AFRL-IF-RS-TR-2003-246
Final Technical Report
October 2003

COMPUTING COMMUNITIES: INFORMATION SURVIVABILITY VIA ADAPTABLE VIRTUALIZATION

New York University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J101

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-246 has been reviewed and is approved for publication.

APPROVED: /s/
PATRICK M. HURLEY
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	OCTOBER 2003	Final Jun 99 – Jun 02

4. TITLE AND SUBTITLE
COMPUTING COMMUNITIES: INFORMATION SURVIVABILITY VIA ADAPTABLE VIRTUALIZATION

5. FUNDING NUMBERS
C - F30602-99-1-0517
PE - 62301E
PR - H556
TA - 00
WU - 01

6. AUTHOR(S)
Zvi M. Kedem, Vijay Karamcheti, and Partha Dasgupta

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
New York University
Department of Computer Science
251 Mercer Street
New York New York 10012-1185

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFGA
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-246

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Patrick M. Hurley/IFGA/(315) 330-3624/ Patrick.Hurley@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***
A Computing Community is a set of mechanism and policies that dynamically aggregates physical resources, information resources, and security resources into an integrated, reliable, adaptable, virtual machine. The computers in a CC run a Virtual Operating System (VOS), which is a software module that executes on top of, and is fully binary-compatible with, the base operating system (Windows NT in our project). CCs are realized through the use of wrapper technology. Employing API-interception, the entire standard operating system is wrapped to create a new, distributed, operating system. Applications that have been previously compiled for the standard Windows NT system will run unchanged on a CC. Several additional technologies augment the VOS. The Application Adaptation system provides self-repair capabilities leading to computation and information survivability. This functionality is provided by the Application Tunability Framework and the Composable, Adaptive Network Services Infrastructure. The Global Resource Manager discovers, allocates, and manages all the resources available to the CC and takes into account the Quality-of-Service requirements of individual applications. This feature is implemented by a ticket based distributed resource sharing agreement and enforcement system. By endowing applications with mobility and dynamic reconfigurability, the CC architecture is uniquely suited for integrating commercial and military systems into flexible, survivable platforms. The CC architecture has been implemented and tested using prototypes and experiments.

14. SUBJECT TERMS
Process Virtualization and Migration, Resource Sharing in Distributed Systems, Application Independent, Application Aware Adaptation, Quality of Service

15. NUMBER OF PAGES
48

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1 Summary

This research report describes the technologies developed and prototypes built that dynamically aggregate physical resources, information resources, and security resources into a *Computing Community* (CC), an integrated, reliable, adaptable, virtual machine. The computers in a CC run a *Virtual Operating System* (vOS), which is a software module that executes on top of, and is *fully binary-compatible* with, the base operating system (Windows NT in our project). The vOS *transparently* endows any standard Windows application with unprecedented capabilities, including the following:

- The application can automatically access all of the CC resources.
- The application can be migrated, without its knowledge; among the CC components, while maintaining all active I/O, open files, and network connections.
- The functionality and performance of the application can be dynamically adapted after faults or intrusions to reflect its current level of importance.

CCs are realized through the use of wrapper technology. By employing an API-interception, the entire standard operating system is wrapped to create a new, distributed operating system. Applications that have been previously compiled for the standard Windows NT system will run unchanged on a CC.

The vOS serves as the CC's core by virtualizing the resources of the original operating system. It dynamically changes the mappings between the resource handles visible to applications and the physical resources controlled by the operating system. The composition of the vOS is modular and consists of an Interceptor, a System Manager, a Software Bus, a Heap Manager, and a Global Policy Manager. The vistualization system handles transparent augmentation of process features, including migration and scheduling.

Several additional technologies augment the vOS. The *Application Adaptation* system provides self-repair capabilities, leading to computation and information survivability. This functionality is provided by the *Application Tunability Framework* and the *Composable, Adaptive Network Services Infrastructure.* The *Global Resource Manager* discovers, allocates, and manages all the resources available to the CC, and takes into account the Quality-of-Service requirements of individual applications. This feature is implemented by a ticket-based distributed resource sharing agreement and enforcement system.

By endowing applications with mobility and dynamic reconfigurability, the CC architecture is uniquely suited for integrating commercial and military systems into flexible, survivable platforms. The CC architecture has been implemented and tested using prototypes and experiments.

2 Introduction

The computing communities project started in 1999 with the concept of building Comptuing Communities (or CC) — a methodology for coalescing a distributed system into a system with no boundaries, augmented with facilities such as virtualization, tenability, adaptation, and survivability. The four major goals stated in the proposal were

1. **The Virtual Operating System** (**vOS**) is a layer of software that non-intrusively operates between the applications and the standard operating system. The vOS presents the standard Windows NT API to the application, but can execute the same API calls differently, thereby extending the OS's power. The vOS essentially decouples the virtual entities required for executing a computation from their mappings to the physical resources in the CC. This layer provides the fundamental mechanisms for building a CC.
2. **The Global Resource Manager** component manages all CC resources, dynamically discovering the availability of new resources, integrating them into the CC, and making them available for use by CC computations. It handles resource requests from other components of the system and satisfies them as per scheduling requirements. It also provides QoS-aware scheduling for applications that require QoS guarantees
3. **The Application Adaptation** component enables the computations to take full advantage of the CC resources and provides self-repair capabilities. It supports computation and information survivability, both in *application-independent* and *application-aware* forms. Application-independent techniques include eager scheduling, two-phase idempotent execution, checkpointing, mirroring, replication, and data dispersal. Application-aware techniques allow computations to become aware of and gracefully adapt themselves to changes in CC resource characteristics
4. **The Information Survival** component is a set of mechanisms and policies embedded in all the above components that provide *accountability*, *communication security*, *information security*, and *recovery from security anomalies*.

Over the years, till the completion of the project (in December 2002), the technologies developed and the goals attained went through expected changes. Goals [1] and [2] resulted in the architecture of a Virtualizing Operating System, with allied technologies of its structure (such as Virtualizing Interceptor, Virtualizing System Manager, and Virtualizing Bus), as well as concepts such as process cloning, persistent sessions, and global system management. Goals [3] and [4] resulted in the development of Application Adaptation Frameworks for adapting tunable application behavior to the underlying physical constraints, CANS – a composable network infrastructure that allows injecting and composing application specific services that rebind network interfaces to adapt to system conditions and to a Resource Sharing System, using agreements and tickets that enforce global resource management.

3 The Computing Community

A Computing Community or CC is defined as:

> *An aggregation of computing and information resources—and even—Computing Communities—all drawn from diverse sources. They are dynamic and hierarchically constructed, self-repairing, and continue delivering adequate, and predictable performance of the key services despite the faults and other imperfections of the execution platform. Effectively, a Computing Community is a single, dynamically changing, virtual multiprocessor system, physically built of many components. The physical network disappears from the view of the computations that run on the Computing Community.*

The goal of CC is to enable a group of computers to work together as a large community of systems, which grows or shrinks based on the dynamic resource requirements through the scheduling and moving of processes, applications and resource allocations between systems.

Computers participating in the CC utilize a standard operating system and run stock applications. The novelty of the CC approach is that it is non-intrusive, causing no application redesign, re-coding, or recompiling. Binary compatibility is assured while adding new services and features such as transparent distribution, global scheduling, fault tolerance, and application adaptation.

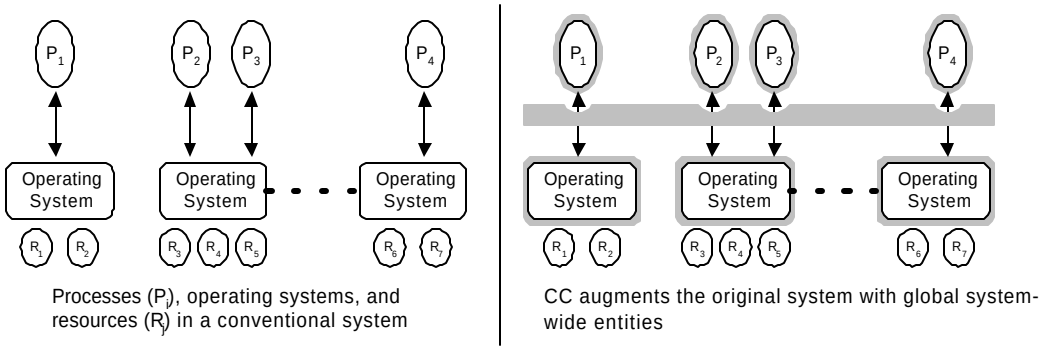


Figure 1: Evolution of the Computing Community

3.1 CC Benefits

Through the separation of applications from their embedded resources, a number of distributed solutions and novel system management opportunities are created – not normally available to legacy applications. These opportunities are defined as features and they form a basis for the benefits of the CC approach. The CC features and their benefits include:

- *Transparent Distributio*n: Resources and application are available anywhere within the community for usage without application knowledge or direct

involvement. A resource is made available and can be relocated closer to an application cluster to improve the locality of the reference. A program or a program cluster is relocated closer to resources that are required in the computation. In an insecure or volatile environment, movement or reconfiguration of resources or applications to more secure or stable conditions may be necessary for system integrity or survival.

- *Global Scheduling*: Resources and applications are scheduled using an active, globally aware facility. Application resource requests for everything from low-level memory, disk space and network access requests through high-level files, objects and abstracted resources are satisfied using a virtual pool of aggregated resources. Whereas the local system manages requests for local resources; these requests are potentially from extra-system sources, handled as if they are local.
- *Fault Tolerance*: Transparent migration establishes the foundation for the replication of state information for executing processes. State storage and restart facilities can be constructed by using this mechanism, which save the application copies to a repository. The repository can be constructed with a variety of levels of redundancy, depending on the environmental volatility and any reliability policy considerations. Using the data in the store, any applications that fail can be restored into a similar environment assigned by the global scheduler.
- *Application Adaptation*: An active component that seeks out and manages its participation in the community. It acts as the local resource manager to requests and integrates community resources. It acts as a watchdog to ensure processing survivability. It attempts to repair any problems in security, resource availability, and performance.
- *Load Balancing*: Using transparent migration and global scheduling in conjunction with policy and performance guidelines, applications are moved to systems with less utilized resources. It is possible to use both individual workstation resources or to utilize process servers that are optimized for load balancing.
- *Synchronization*: Synchronization controls utilizing native APIs combined with vOS system controls are used to create cross system synchronization of events, objects, and actions. Latency and failure modes are always a concern in this type of system; however, by using priority messaging in combination with active resource consistency, latency can be managed or controlled for most applications.
- *Binary compatibility*: The application and the host operating system are not aware of the presence of the community or of the virtualization implementation. Thus, no change is required to any of the application's or system's binaries. This feature alone results in tremendous savings to the implementation of the community into a stock environment.
- *Persistent Sessions*: By using any existing or virtualized extensions to security policy in conjunction with transparent migration, user sessions are made to be persistent during a session and beyond the session termination. This allows users to perform either a partial or full logon during an existing session, or to restore a

saved session after logoff. In addition, fully or partially shared sessions are available between groups of users.

- *Pre-forma Scheduling*: Hints and advisories of resource requirements along with policy guidelines are used to pre-schedule resources. This provides overlaps that boost performance by reducing scheduling and waiting time.
- *Policy Profiles*: Computing communities operate using policy profiles to guide and control community and individual decisions, both locally and globally. Local policies control participation in the community by the local system applications and resources, and they establish guides for performance and security. Global policies guide migration, scheduling and adaptation practices.
- *CC Awareness*: Applications that are developed to be CC aware have direct availability of the community facilities and resources to enhance their operation. This includes participating in migration, scheduling, fault detection, adaptation and persistence decisions, which may include direct usage of CC resources. The awareness also includes direct community cooperation through the exposure of APIs, abstract resources and objects that participate in the community.

4 Virtualizing Operating System

To achieve the core of the Computing Community, the key technique is the creation of a virtualizing Operating System or vOS. The main theme in the vOS is "virtualization," which is the decoupling of the application process from its physical environment. That is, a process runs on a "virtual processor" with connections to a virtual screen and a virtual keyboard, using virtual files, virtual network connections, and other virtual resources. The vOS has the ability to change the connections of the virtual resources to real resources, at any point in time, without any support from the application.

The vOS is a centrally managed, distributed, multi-system facility that provides key global coordination and control services, see Figure 2. To ensure that the vOS is scalable and to eliminate or overcome any problems of vOS fault tolerance, multiple vOSs operate as peers and coordinate activities between their respective domains. Independent state repositories or cooperative storage between the vOSs allow for fault tolerance within the vOS community, refer to section 4.5.3 for additional details.

The vOS can be located anywhere in the network and can perform global functions. It can work with the virtualizing EXecutive (vEX), a system command and control component residing on each participating system. The vEX is a Windows NT service acting as the vOS's local agent and proxy, refer to section 4.5.2 for additional details.

The vEX uses local workstation resources in combination with a wrapper tool and the virtualizing INterceptor (vIN) to capture and manage the workstation processes that participate in the CC. The vIN is used by the vEX to initiate and capture process data from the workstation applications.

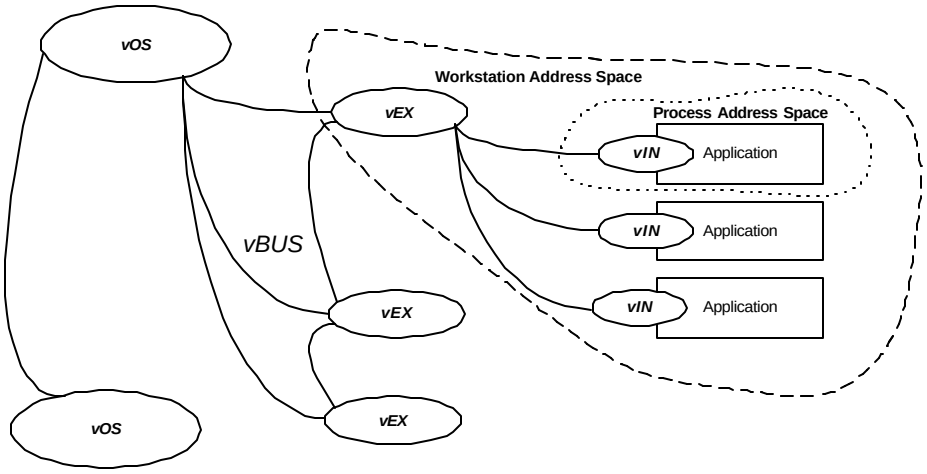


Figure 2: vOS System hierarchy

4.1 API/DLL Interception

Adding or extending the architecture or capabilities of a workstation operating system normally involves modifications to the underlying system components, in conjunction with adjustments to the APIs. Two primary forms of adjustment are possible:changes to the existing APIs or additions of new APIs. If the signature of an existing API is altered, each application must be examined and modified using the original API. This type of change creates ripples throughout the entire application system, and is a very expensive proposition. For this reason, published APIs are seldom modified.

One approach to modifying the existing APIs is through the addition of parameter values that extend the meaning of the existing parameters. As long as the changes are not signature changes, or parameter semantic re-interpretations, no change is required to the existing applications. If there are semantic changes, then the applications must be examined and changed.

Another approach to enhancing a system is through the wholesale addition of new APIs. Existing applications incorporate the new API capabilities through reengineering, while new applications have the choice of completely ignoring legacy APIs in favor of the new APIs. However, this approach does little to extend or add functionality to the existing legacy systems without changing them. In effect, a distinct resistance or even a barricade to architectural change is created and only new applications use the new APIs.

The Windows NT system provides a late binding API architecture through Dynamic Link Libraries (DLLs). Late binding creates the opportunity to insert a middleware component into the system that examines, modulates, or replaces the API calls. The insertion of the middleware component creates the API interception mechanism.

The API interception forms the basis for the injection of system functionality in CC. By inserting code between the application's API call and the system's API, a new functionality is introduced.

When the application is loaded in the Windows NT DLL scheme, the API references are resolved to a table of addresses in the user space, called the Import Address Table (IAT), and are filled in at run time. The DLL contains a list of exported addresses that are used to populate the table. Using an indirect pointer, the application jumps to the API entry point within the DLL. By modifying the addresses in the IAT, the application call is redirected to an alternate API entry point, as shown in Figure 3.

The new API code acts as a wrapper around the existing application code and has access to all the data explicitly passed and returned by the API call. A new functionality is added through the manipulation of the data in these calls, and state information is captured for subsequent usage.

The API Hooking mechanism provides the basic mechanism for the API interception. By using the constructor instantiation mechanisms, inherent in C++ object classes, API addresses in the import modules of every DLL and the IAT entry within the user application are replaced by the address of the newly defined hooking procedure. This technique hooks the API references within all of the system DLLs mapped in the process address space, which is then exploited in the Process Interdiction described above. The Richter API Hooking procedure also provides a method to restore the original. API hooked addresses when hooking is complete. With the addition of modifications to allow specific modules to be hooked rather than all the references, this approach proves to be the most effective and efficient method of performing process hooking.

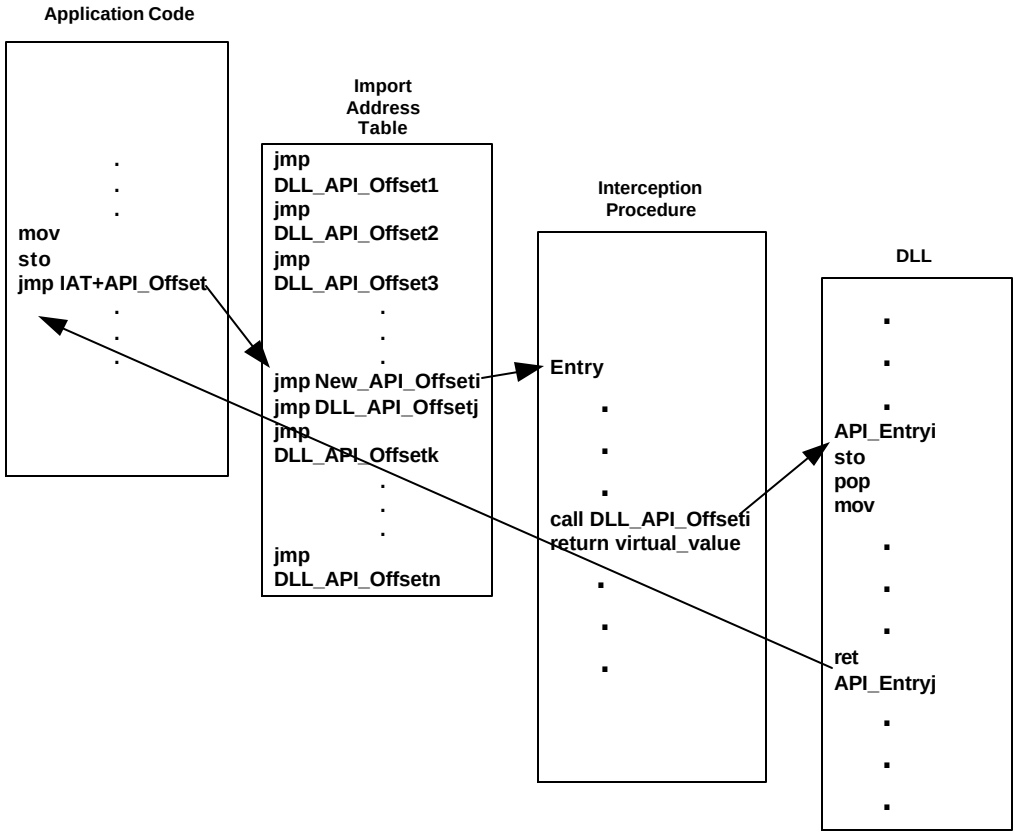


Figure 3: API Hooking Model

4.2 Process Interdiction

The Richter API Hooking technique replaces all references to a target API in every module found within the given process space. By hooking the CreateProcess function, specific applications that are launched are detected and can be groomed to become part of the vOS. This approach is referred to as Process Interdiction and provides the basis for the dynamic operation of the system.

The Windows desktop is an application known as explorer.exe. Explorer is launched and executes continually as the standard user interface for Windows. When the mouse is used to launch an application, the explorer creates the new process based on the user selection. With the explorer modules hooked, the process is interdicted or caught before it begins execution. At this point in the process initiation, various actions occur. The initial action is to validate with the vEX that the interdicted process is in the Watch List, which is the list of modules that execute under the vOS control. If the launched application is in the Watch List, then it is created in a suspended mode, otherwise the normal process creation activities continue undisturbed. Once suspended, the vEX injects the application with the vIN, which in turn installs the process hooks causing

virtualization. When the vIN signals the completion of its setup tasks, the process is resumed, and the application appears to the user.

4.3 Resource Virtualization

The Windows NT system is architected to use handles as references to almost every component and resource of the system. The files, networks and communications, processes, threads, fibers, events, windows, menus, submenus and edit buffers are just a few of the resources that have handles associated with them. These handles are unique to each system in that they are not deterministically regenerated for a given application and therefore they are not system interchangeable.

To virtualize applications and resources, there is a need to create and map new handles and replace references within API calls between systems (see figure 4). Virtual handles allow each API to function correctly on the local system and thus form the basis for abstracting resources from specific system instances. Although handles can be extracted during system operation through standard API references, they are best captured and virtualized when they are created by system calls.

Handles normally consist of a 32-bit value. To aid in tracking and debugging, the handle is encoded with an origination code. The code uses a flag in the first nibble to indicate that the handle is virtual. Currently handles do not use this bit position in the 32 bit handle value. Future implementations can include identifiers for the source machine, process, thread, and handle type. This information could be useful for tracing or debugging a migrated process, especially after several migration iterations.

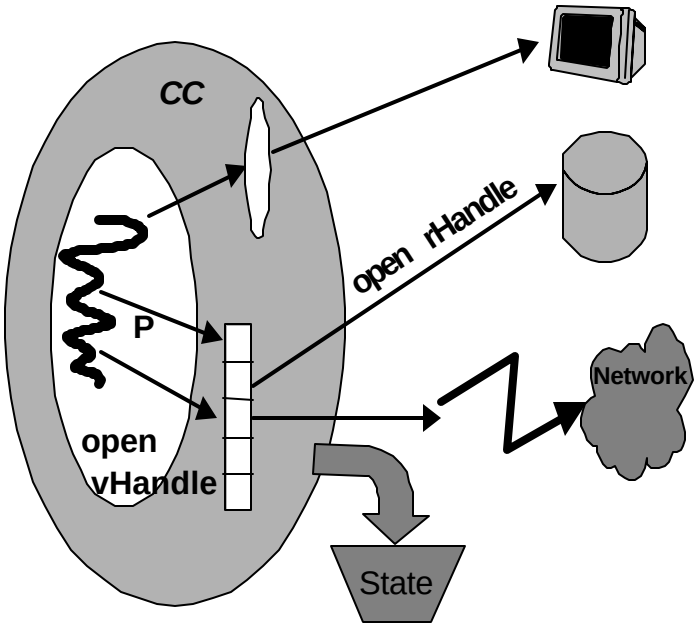


Figure 4: Handle virtualization

Windows NT process state information is for the most part the same as most of the currently available systems, such as Linux. The hardware system consists of registers, stacks, the dynamic data of the individual process, the system state which may affect the process context, and the application's current semantic context. Virtualizing processes and resources include various captured, replicated and restored contextual components. The components of interest include:

- *Program Text*: The executable code, which includes EXE and DLL files, is immutable in the migration context. It is available from the shared file system and does not need to be explicitly migrated. It is loaded to the same address on the target machine by using the system call to create a process, a thread, or a fiber. It is initially created in a suspended state, so that the state and context information can be restored from the source machine. Once restored, the new thread is restarted. In the case where the application is not available, policy and profile guidelines will be used to determine whether to copy the text to the target system. Before copying the text, the nature of the application installation on the Windows NT system needs to be taken into consideration. The system registry settings, control files and application DLLs are sometimes used to define and support the application's function.
- *Process, Threads, and Fibers*: Windows NT has three forms of organizing and executing application program codes. The first and highest level form is the process. It is a code that is included in one of the memory partitions that the operating system schedules and keeps separated from other process environments by using hardware and security protection. The second organization of NT is threads. They are system-scheduled processes that run within a single memory partition with no hardware protection within the partition but full protection from other partitions. In this respect, a process is a thread. The third organization of NT is fibers, which are the applications managed and the scheduled processes that are unknown to the operating system scheduler, but are protected by the memory process partition.

Virtualization of programs containing these three forms requires that the process structure be mapped, connections to resources be virtualized, and state information be collected and maintained. It is best to perform the data collection at the onset of the application through the process of a loader and wrapper operation. This approach allows each construct to be identified as it is created rather than attempting to examine the environment de facto and potentially overlooking a detail.

4.4 vOS Architecture

The virtualizing Operating System (vOS) is an integrated middleware application management system designed to unobtrusively utilize existing system constructs and components. Using these standard functions, it provides services that intercept and virtualize existing applications without altering the application’s coding in any way. Once virtualized, the vOS extends the capabilities of the application. The primary extension is the ability to migrate a virtualized application within the vOS environment.

The structure chosen is that of a tree with a single controlling element at the base, communicating with multiple individual systems through a module or an agent that resides on each system, which in turn communicates with the module assigned to manage an individual application. The guiding philosophy is one of system independence. This is accomplished by creating individual components that do not marginally depend on the other components. This allows for systems to fail and be either restarted or replaced dynamically.

The main architectural components of the vOS are the Interceptor (vIN), Executive (vEX), System Manager (vSM), Communications vBUS, and Heap (vHEAP). The following sections provide a review of the vOS five main components

Virtualizing Interceptor

The virtualizing Interception (vIN) provides the connection between the vEX, the Operating System, and the target application. The vEX initiates the vIN when an application is created. The vIN is injected into the application's process space, while the application is in a suspended mode. Once the vIN is loaded, several threads are established by the vIN within the process state to handle: the initialization, intersystem communications, command, and control. The module that provides the support for the API interception is integrated into the existing program thread during the interception process.

Each vIN communicates with the local system executive using a combined common and unique shared memory data structure. Through this mechanism, signaling and simple inter system data transfers, such as command and control directives, are performed. For more complex data transfers as well as extra system communication, named system pipes are utilized. There is a one-to-one relationship between a vIN instantiation and an application that participates in the vOS.

Earlier testing and examination of the state information of a process have led us to the conclusion that capturing API information during the initialization calls is necessary in order to understand the complete API state and structure. Otherwise, certain types of specific state information can not be determined by using or by relying on the existing API references.

Virtualization tables are created through API interceptions. These vTables are built and managed by a vIN module using a set of specialized APIs included in a standardized migration library that is coupled with a user defined API virtualization module. API calls that use or return handles always use a virtual handle that is created and maintained by the migration modules. When migrating a process, the migration modules provide the mechanisms to recreate and reassign the new virtual handles during the restoration process. The virtual handle references are forwarded, as part of the application's state information, when the process is cloned.

After the vIN is established/created by the vEX, it acts as an autonomous component. The vEX performs the API capturing, marshalling, and unmarshalling without requiring any external involvement. In the case of a problem or catastrophe, the vIN can be deployed to perform any autonomous functions that might be required for providing some level of self-preservation, such as reinstantiating a new vEX in case that the existing vEX

becomes unavailable, or if system integrity is effected adversely. vEX is also able to checkpoint itself and the associated process for later retrieval and redeployment.

Virtualizing Executive

Each system participating in the vOS contains a local system executive (vEX) process. The vEX provides system level coordination and control. It ensures that each individual process is loaded by using the vIN, and it then acts as the initial IO connection point for all the initial API connection traffic. All commands and control originate either at the vEX or at the vSM. Changes in process, security or resource state directed by the controlling vSM or detected locally by the vIN or vEX are managed by vEX. When the vSM issues a command, the vEX acts as the pass through to the vIN.

The vEX is a multithreaded process residing within its own process space. It is an NT service that runs in the secure context provided by an NT service. It is capable of running as a regular application with no special requirements other than security. The vBUS provides the intra system delivery of command and control, local user interface, resource, migration, policy and failure management requests and exchanges.

The vEX is autonomous and quasi-persistent. If a local system failure occurs, the vEX can checkpoint its own state, as well as the states of the vIN's and the application, and can migrate or be migrated elsewhere within the vOS. It is also responsible for maintaining the current and remote resource and policy state for the local system. It performs this role by exchanging information periodically with the vINs and the vSMs.

Communication between the vEX and each vIN is facilitated through the use of an inter system global shared memory. The shared memory contains signals and command controls for initiating communications between each vIN that is instantiated and the vEX. Once initiated, simple signals, in conjunction with a standardized set of commands, are utilized.

The user interface for each machine is presented and managed by the vEX. Information on system state and local system command and control is provided. In addition, a console messaging display or log is integrated into the interface. The log provides a central location for vEX and vIN state action messages to be displayed during the system operation. The interface consists of a window partitioned so that the various pieces of information can be easily viewed and the commands will only require a simple "button" press.

Virtualizing System Manager

The System Manager (vSM) is the control component of the virtualizing Operating System (vOS). It is a multithreaded process residing on any system within the scope of the vOS, including a system that contains a vEX. It is a self-migratable component that carries the roll of a central and primary controlling agent, as well as the information source for the system. The resource list and status for each component of the system is persistently maintained by the vSM. Each vEX communicates with the vOS to obtain/get information about the other vEXs and system resources.

Command and control functions are performed by the vSM for policy, process migration, recovery control, and security. The primary direction of control is towards the vEX. It

directs activity based on an overall system view, acquired through operator control or system input heuristics.

The vSM is the primary interface with the user; for system status reporting, command and control, system initialization, and shutdown. The user interface is a windowed pseudo real-time interface that reports the current state and activities of the vEXs and vINs, as well as the system resources. Similar to the vEX, the vSM contains a message console for vSM and general vOS status messages.

The vSM utilizes the vBUS to communicate bulk data with the individual vEXs. For simple, aggregate control of the vEXs, a multicast module is used. Multicast allows for a minimal state management model to be utilized by the vSM. Brief command messages are utilized with responses that originate new commands across the vBUS by using the vEX.

An advertising scheme is used by the vEX to find a vSM. The vSM sends out a multicast message periodically to indicate that it exists. vEX listen to this message, then sends a registration request to the vSM that the vSM acknowledges. This approach allows for relatively frequent advertising without using a large amount of bandwidth.

Virtualizing BUS

The vOS is a logical system constructed from multiple individual workstation components. Its overall performance depends upon effective interaction between the workstation instances. Efficient and timely sharing and exchange of information, status and messages flows between and throughout the vOS environment are defined in terms of singular, group and universal relationships. This creates a requirement for the use of a multi-shared , distributed communication methodology. The best architectural concept to fit this requirement is that of a BUS. To this end, this research includes the development of a virtualizing BUS, in support of the overall system operation but within the context of the Windows NT and general internetwork environment.

The virtualizing BUS (vBUS) is architecturally similar to the hardware BUS, in terms of the approach to the logical presentation and control of data. It has a simple and efficient API that is used for receiving and delivering signals and messages. Signals and messages are delivered to one or many listeners, depending upon addressing controls. The types of message ordering can vary from none to strict, with delivery guarantees ranging from best effort through confirmed delivery depending upon selected preferences. Message types range from discrete to streaming, depending upon application data use and movement requirements.

The architecture of the vBUS utilizes the existing transport facilities: IP, TCP, UDP, and any inherently available capabilities. With the introduction of RFC based multicast and broadcast protocol support, new opportunities are available to use facilities such as the Multicast Backbone technology.

To support asynchronous message events to multiple resources, a callback approach will be implemented. To use this callback approach, an application, such as the vIN, makes a function call to acquire access to the BUS, thus creating a registration and insertion into the BUS. This effectively adds the requesting function to a table or list of functions to

call when a message arrives. Message classes and identifiers are used to activate the registered processes procedure.

Virtualizing HEAP

The memory allocation mechanisms of Windows NT and 2000 use, the VirtualAlloc()API [MS01] function. All calls made to various current and legacy API references, such as HeapAlloc(), LocalAlloc() and GlobalAlloc(), ultimately resolve to a call to VirtualAlloc(). Windows memory allocation is concerned with the optimal use of memory and not with deterministic allocations. For the implementation of the Minimal State migration methodology described earlier, deterministic heap management is required. The virtualizing Heap (vHEAP) provides this support.

Providing deterministic operation of the heap means that the location of the heap allocations must be consistent between each execution of a process or thread. To accomplish this, there is a need(/or it's necessary) to create some rigidity of allocation. There are two aspects to this rigidity: the first is the duplication of location and the second is the management of a location change during execution.

Memory allocations for applications can range dynamically throughout the Windows address space. However, during normal operations on the same machine, they tend to be allocated in the same order and in the same locations until some aspect of the machine memory is changed. An example of a change affecting memory is the addition or removal of a new device driver. Another example is the variance in system modules due to versions differences.

To ensure that memory is allocated at the same location not only on the same machine, but also between various machines, it is necessary to allocate a fixed memory location in order to ensure that no other application or system module allocates within the same space. The order of the memory allocations for most applications does not vary during the application execution. This means that the order should maintain itself without requiring any direct system involvement.

Memory releases, consolidations and reallocations must not cause memory to be reused, no matter how wasteful this would seem. Normal reallocations can reuse areas that were previously freed, thus causing memory addresses over the course of the program execution to be potentially duplicated. Since virtual memory addresses are captured, reallocated and potentially replayed during process migration using the Minimal State approach, it is important that the addresses remain unique so that the data is not replaced and the data area references remain identical.

Memory allocation for the Full State and Full Distributed States approaches do not require the use of vHEAP. They use a copy and recreate a memory approach that replaces all memory references in their pre-existing locations.

4.5 The vOS Prototype

The vOS system is implemented using the components described above. In addition, several components are added to provide support for process migration and command and control communication between the system components. The vOS is implemented using the philosophy of Self-Managing Requests (SMR). SMR, in its simplest form, means

that the state and needs of a component are best known and managed by the component itself. This approach works somewhat in the direction of a stateless model, without fully abandoning the advantages of sharing minimal state information.

A complete abstraction philosophy is enjoined for the vOS. Every API call for the native operating system, in this case Windows NT, is replaced with a call to a vOS library version of the call. In most cases, there is a one-to-one match, however, some calls are, in some cases, combined or simplified for specific purposes. Using abstraction will allow a more ready porting of the system to newer versions of Windows as well as to different operating systems.

The vOS system is implemented using a layered methodology. The main executable modules are the vSM and the vEX. The remainder of the system consists of dynamic link libraries, loaded as and where required, such as the vIN. The main abstraction functionality is within the vLIB and vBUS; this includes replacements for virtually every API reference made within the system. MultiCast provides the general command and control communications between the vSM and vEX. The specialty work modules, InjLib, HookAPI, APIHook and HookAPILib, provide the support for the API interdiction. HookDefault and MigrationLib are the main interception logic modules installed by the specialty work modules. vHEAP provides the memory management for the Minimal State migration methodology

In support of the migration process, the data is mapped into an XML based syntax, defined by the Migration Markup Language (MML). The MML allows for a high degree of flexibility in implementing additional migration methodologies as well as making further state information easier to integrate into the current system.

5 Application Adaptation

Increased platform heterogeneity and varying resource availability in distributed systems motivate the design of *resource-aware* applications, which provide the desired performance level by continuously adapting their behavior to changing resource characteristics. As part of the Computing Communities project, we developed an application-independent adaptation framework that simplifies the design of resource-aware applications. This framework eliminates the need for adaptation decisions to be explicitly programmed into the application by relying on two novel components: (1) a *tunability interface,* which exposes adaptation choices in the form of alternate application configurations while encapsulating core application functionality; and (2) a *virtual execution environment,* which emulates application execution under diverse resource availability enabling offline collection of information about the resulting behavior (see Figure 5). Together, these components make automatic runtime decisions on *when* to adapt by continuously monitoring resource conditions and application progress, and *how* to adapt by dynamically choosing the most appropriate application configuration for the prescribed user preference. The tunability interface exposes choices in application execution paths allowing the enumeration of alternate application configurations, while the virtual execution environment permits the automatic and offline generation of profiles

that indicate/describe how each configuration behaves under different resource conditions.

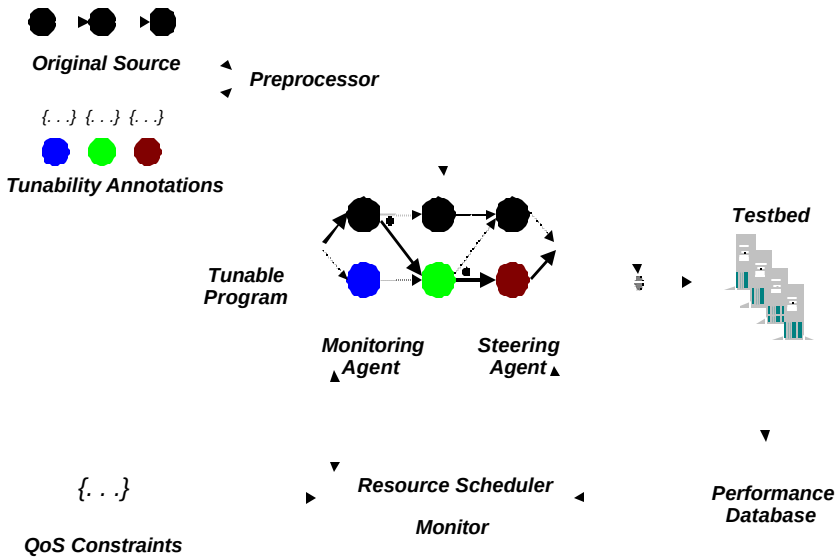


Figure 5: Application Tunability Framework

Together, the two mechanisms enable (or facilitate) the development of a runtime adaptation system, which continuously monitors the resource conditions and application progress (in terms of user preferences of QoS metrics). They also automatically determine both *when* adaptation should be performed and *how* the application should be modified (i. e., which of its configurations should be chosen) based on application profiles. The automation of the adaptation procedure ensures that the application achieves the desired level of performance, while minimizing the programmer's involvement in performance-related considerations. This framework has been implemented in the context of distributed and parallel applications running on Windows NT platforms. The tunability interface is exposed using language level annotations that allow a preprocessor to generate monitoring and steering agents, as well as control messages, thus permitting external access to and control of the application execution. The virtual execution environment is implemented using a novel technique, called the API interception. The latter emulates different availability of system resources by continuously monitoring and controlling application requests for system resources. Runtime adaptation takes advantage of the steering and monitoring agents. This framework has been evaluated using a distributed interactive image visualization application and parallel image processing application. Our experiments demonstrate that by starting from a natural specification of alternate application behaviors, it is feasible to model application behavior using the virtual execution environment in sufficient detail to automatically adapt the application at run time to changes in CPU load, network bandwidth, data arrival pattern, and other environmental characteristics. Each application adapts by choosing a

configuration that employs a different algorithm, or one where the execution sequence is modified to satisfy user preferences of output quality (e. g., image resolution) and timeliness.

5.1 Example Applications

Many applications exhibit the flexibility to choose different paths of execution at run time, thus trading off resource requirements over several dimensions, including output qualities, resource types, or computational stages. These applications are *tunable,* in the sense that, their flexibility permits adaptation by switching over to a different execution path at run time, ensuring user preferred performance levels despite the changes in the characteristics of the execution environment. In this section, we describe two example applications: 1) a distributed image visualization application and, 2) a parallel image processing application to examine how they permit runtime configuration and adaptation.

Active Visualization

The Active Visualization application is a client server application for interactively viewing, at the client side, large images stored in the server. Figure 6 (left) shows the overall structure of the client and server processes. The application uses multi-resolution and progressive transmission techniques to improve performance. First, images are stored at the server as wavelet coefficients, thus enabling the construction of images at different levels of resolution. Second, the server employs progressive transmission to improve response time. Based upon the initial specification of the highest resolution required by the client, the server constructs a pyramid of images, ranging from the finest to the coarsest resolution. The server uses this pyramid to transmit an area of the image that corresponds to the user's fovea (focus of interest), starting from the coarsest resolution and progressing up to the user preferred resolution. If the user's fovea does not change, the client requests the server to send an incremental region surrounding the fovea, thus ensuring the eventual transmission of the entire image at the highest required resolution. Finally, the image data is compressed to reduce bandwidth requirements.

This application promotes adaptation by permitting the association of different values with parameters, such as preferred resolution level, size of the foveal region, and choice of the compression method. The resultant executions trade off resource requirements for application performance: a low resolution implies less resource requirements; different compression methods require different CPU and network resources; and a small fovea size leads to a quicker response for getting the foveal region, but a larger number of rounds is needed/required to transmit the whole image.

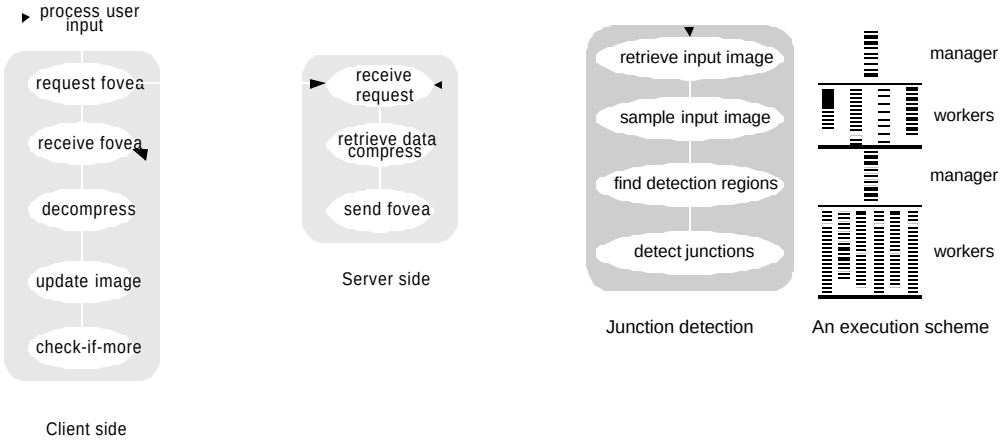


Figure 6: Structure of the Active Visualization (left) and Junction Detection (right) applications.

Junction Detection

The Junction Detection application is a parallel image processing application running on the Calypso system – an adaptive parallel processing system which views computations as consisting of several parallel tasks inserted into a sequential program. Each parallel task can run on a changing set of *worker* machines and is responsible for performing the computationally intensive work. The sequential code, on the other hand, runs on a dedicated *manager* machine and is responsible for the high level control flow and I/O. These parallel tasks can be executed multiple times (with possibly some partial executions), with exactly once semantics. Within a parallel step, Calypso supports CREW (concurrent read, exclusive write) semantics to shared data structures, with updates visible only at the end of the step.

Junction Detection is the core component of several image-processing applications and detects distinguished sets of pixels in an image where the intensity or color changes abruptly. Figure 6 (right) shows the overall structure of the computation, which consists of a driver routine initiating a component for processing an individual image. The per image component consists of three parallel steps. The first step samples a subset of the pixels in parallel and performs a quick test to determine whether or not the tested pixel is of interest. A pixel is of interest if the difference among intensities/colors of its neighbor pixels is beyond a threshold. The second step draws a region of interest around a cluster of interesting pixels. The region is essentially a convex hull containing at least a certain number of interesting pixels, in close proximity. Finally, the third step runs a compute intensive algorithm for every pixel in the region of interest. This application permits adaptation by exploiting different numbers of worker machines for their parallel tasks and selecting appropriate sampling schemes for images with different numbers of interesting objects. The resultant executions trade off resource utilization for application performance: allocating more worker machines to a single application may speed up its execution but degrade the overall system performance; and the arrival of images of

different characteristics requires different sampling schemes to best utilize the system resources.

5.2 Application Structuring

Adaptation requires the existence of multiple ways of executing an application, or *alternate configurations,* which exhibit different resource utilization profiles. These multiple profiles provide runtime flexibility, thus permitting the choice of an alternate configuration that is better suited to the matching user preferences than the available system resources. The alternate configurations can be built out of different application modules or from the same module that exhibits different behaviors under the influence of some control parameters. For instance, a video compression component can trade off compression ratio against image quality either within the context of a single algorithm, or conceivably by switching between different algorithms to allow a bigger tradeoff range. Although the transition among different configurations can be achieved completely at the application level, it can be controlled more effectively at the system level, without requiring extensive programmer involvement. However, this requires some form of application that provides an independent way of enumerating the different configurations, as well as the mechanisms that effect transitions among the various alternatives. We meet these requirements by relying on a structuring concept, called the application *tunability interface*. The tunability interface provides language support to express the availability of multiple execution paths for the application, as well as the means by which application progress can be monitored and influenced (by switching to a different path).

Tunability Interface

In our framework, applications are assumed to be built up from multiple components that are linked together by using standard control flow mechanisms. Different application configurations correspond to different behaviors for these components, exported in terms of the tunability interface. In general, the tunability interface of a component provides five kinds of information, that are described below. The sample code fragments correspond to our current implementation of this interface, which is based on eXtensible Markup Language (XML) annotations to C/ C++ source.

Control parameters and their value ranges provide the "knobs" by which component behavior can be influenced. Setting different values to the control parameters results in the component following a different execution path. These parameters are usually local variables inside a program module; but are promoted as part of the tunability interface by giving them an external name and listing them in the `control param` construct. For example,

```
<control_param>
  <param type="int" externname="compress"
     localname="c" onchange="func" />
</control_param>
```

identifies a local variable, `c` of type `int`, and promote it as a control parameter `compress`. The local variables and control parameters exchange their values when the application execution enters or leaves the corresponding module. The value of the control parameters could also be changed either through an external notification from a system

scheduler or through another application instance. In such cases, the function `func` (as specified by the `onchange` attribute) is invoked, given the old value and new value, thus allowing application specific actions.

Execution environment specifies the system entities (hosts and network links) on which the application component executes. Each system entity, in turn, encapsulates several resources that affect component behavior. For instance, a host is characterized by its CPU, memory, and network resources. The execution environments are specified using the `resource param` construct. For example,

```
<resource_param>
  <host name="client" local="true" />
  <resc type="networkBW" name="net" host="client" />
</resource_param>
```

specifies a host client with one resource parameter of interest, as well as the network bandwidth (referred to by the name `net`) in the execution environment. The system monitors the specified resources to detect any changes in their characteristics, potentially requiring adaptation of the application components.

QoS metrics and the expressions for evaluating them specify the component metrics of interest as well as the application provided means of computing them. Although QoS metrics are application dependent, they are assumed to be comparable and have min/ max values. The construct

```
<QoS_param>
  <qos type="double" dir="dec"
     externname="response_time"
     localname="response" />
</QoS_param>
```

specifies one QoS metric `response time` of type `double`, mapped to a local variable `response`. Application performance improves whenever the value of this parameter decreases. QoS metrics are evaluated in the `QoS monitor` constructs that are embedded in the source code.

Tunable modules and intermodule control flow set up the alternate execution paths. Each module is specified by a `component` construct. The application execution paths are specified by associating guard expressions of control parameters with each component and specifying intercomponent control flows using sequencing, conditional, and looping constructs. The following construct

```
<component name="module\{compress\}\{...\}" />
  <condition> expr </condition>
  other constructs ...
  application source code ...
</component>
```

describes a component with guarded expression `expr` and a name that is comprised of characters and control parameters. The control parameters associated with the component name are evaluated as namevalue pairs when the component is instantiated at run time, and serves as a handle for referring to a specific application configuration. Inside component constructs, other constructs could be defined, such as control parameters, execution environment, QoS metrics, and nested components.

Transition functions encapsulate application specific actions, such as updates to local variables and/ or control messages sent to other components, which are required upon a change in the component configuration. Compared to the `component` construct that allows adaptation upon module entry, transition functions enable adaptation inside a component. The construct

```
<transition from="oldctl" to="newctl">
   application-specific code ...
</transition>
```

defines a transition point with two sets of control parameters, `oldctl` and `newctl`, where the `newctl` parameter identifies the configuration suggested by the system, based on the current resource conditions. The application specific transition function could examine these two sets and decide how to proceed, possibly overriding the system's decision. At the end of executing this function, the control parameters reflect the `newctl` values.

A preprocessor implemented using an XML parser processes the above interface, in the form of source level annotations. The control parameters, QoS metrics, and execution environment definitions are converted into data structures in the source language (e. g., structure/class declarations in C/C++), and are accessible to other annotation constructs. Different execution paths are achieved by expanding the tunable component constructs into conditional statements involving the associated guard expressions. Changes in component configurations are detected by inline checking codes. In addition to the tunable versions of each component, the preprocessor also generates *monitoring* and *steering* agents that monitor resource availability and control application execution, respectively. These agents interact with the application components by accessing the tunability data structures using shared memory.

Besides generating the application components required for flexible runtime execution, the preprocessor also generates configuration files that serve as input to both the virtual execution environment and the external resource scheduler. The configuration files identify the application components and their control parameters, as well as the execution environments and the QoS metrics. The availability of this information allows an external source, either a driver program or a system level scheduler, to interact with a running application and perform a variety of tasks, ranging from passive measurement (e. g., recording the performance achieved by a particular configuration, under a given resource condition) to more active control (e. g., changing the behavior of application components).

Example. Figure 7 shows the adaptation structure of the client side of the Active Visualization application, as it appears when the tunability interface is used. The progressive refinement component is controlled using three control variables – `fovea`, `resolution`, and `compress` – which influence the foveal region size, the image resolution, and the compression method affecting the behavior of the `request`, `decompress`, and `check-if-more` components. Two QoS metrics – `response time` and `transmission time` – measure the performance of this component. Note that the control parameter `resolution` is performed at the same time that the QoS metric for this application is performed. Finally, the transition function updates the control

parameters in the server instance of the application whenever there is a change in the compression method. This update takes the form of a control message sent to the server.

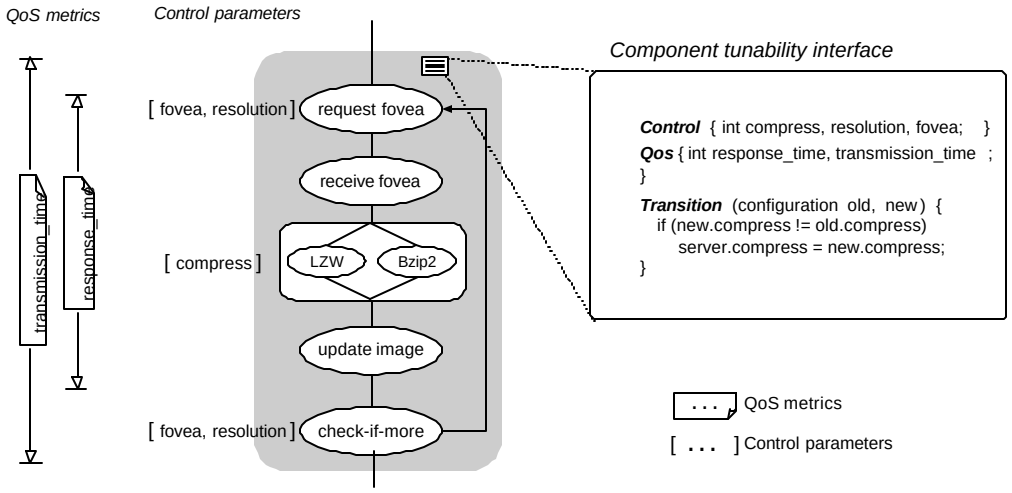


Figure 7: Exporting component tunability for the client side of the Active Visualization application.

5.3 Modeling Application Behavior using a Virtual Execution Environment

The explicit specifications of the application tunability facilitate the development of a model of behavior for each application configurations that is expressed as the mapping from control parameter (application input can be viewed as an additional control parameter), and resource conditions to application-specific quality metrics. Due to the difficulty encountered in obtaining an analytic expression, a profile-based modeling is used to approximate this mapping.

We obtain the profiles by using *virtual execution environment*s. These environments run on top of a static distributed system, and can be configured to accurately emulate a variety of resource availability scenarios. Our implementation of these environments relies on standard mechanisms, available in most current-day operating systems, and the API interception technique, and supports the constrained execution of unmodified applications.

Given the nature of these environments, obtaining profile-based models of configuration behavior is straightforward. A driver script executes each configuration repeatedly, thus setting up the virtual execution environment to sample different resource conditions. A separate tool analyzes the output quality measures to determine any configurations and regions of the resource space that might require additional samples. The output of this modeling step is a *performance database* that records information about a maximal subset of the configurations which represents the resource profile of this application (informally defined as configurations that outperform other configurations under at least one resource situation). These measurements are interpolated to get performance curves that summarize the configuration performance.

The virtual execution environment is implemented by effectively creating a "sandbox" around an application, which constrains the application utilization of system resources, such as the CPU, memory, disk, and network. Our specific implementation on the Windows NT platform relies on the ability to inject arbitrary codes into a running application, using a technique known as the API interception. This code continually monitors application requests for operating system resources and estimates a "progress" metric. Upon detecting that the application has received adequate service for the virtual environment, the injected code actively *controls* any future application executions by relying on the OS mechanisms, to ensure compliance with specified limits. This technique, which has been demonstrated in the context of three representative resources — CPU, memory, and network, allows accurate control over resource availability to applications on commodity OSes without any modification to either the OS or the application source code.

Example. Figure 8 shows some example performance profiles for the Active Visualization application that illustrate how the transmission time and response time metrics vary with different amounts of CPU, shared for different values of the fovea size parameter.

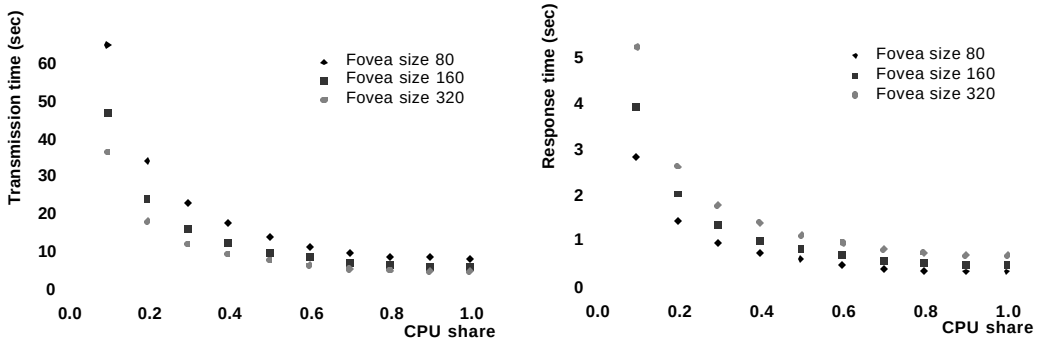


Figure 8: Performance Profiles for the Active Visualization application, obtained using the Virtual Execution Environment.

5.4 Run-time Adaptation

As described earlier, the program annotated with tunability interface specifications is converted by a pre-processor into an executable form; the latter includes application modules that make up the different paths, as well as the steering and monitoring agents used for runtime adaptation.

At run time, an appropriate application configuration is chosen to satisfy user preference constraints, and this selection is dynamically updated as the resource availability changes. Such configurations and adaptations are facilitated by interactions between three components: (1) an *application specific monitoring agent* that monitors the resource characteristics that are important to the application, as well as to the application progress; (2) a *resource scheduler* that correlates the observed resource characteristics and the

user's preferences with the performance models, stored in the performance database; and (3) a *steering agent* that performs the actual reconfiguration.

Each user preference constraint is expressed as a value range, on a subset of output quality metric, and is accompanied with an objective function that needs to be optimized. For simplicity, we assume a relatively restricted form of this function; maximizing or minimizing a single quality metric. When these constraints are considered together with the measured resource characteristics, they restrict the suitable set of application configurations. The *resource scheduler* picks the constraint that best satisfies the objective function. Multiple user preference constraints can be specified. The system examines them in decreasing order of preference; in the case that one request cannot be satisfied due to inadequate resources, the system attempts to fulfill the next preferred constraint.

The *monitoring agent* continually observes the application progress and estimates the fraction of resources of interest that are available for use by the application. To do this, the agent relies on generic progress metrics (e. g., comparing allotted CPU time with the wall clock time after factoring in periods where the application is waiting) and a system database that provides information about the maximum capacities of the system resources (such as CPU speed and the number of physical memory pages). Our experiments show that such monitoring incurs negligible overhead to application execution even when it is performed in very fine-grained time slots. The *steering agent* listens to the control messages and is responsible for switching application configurations. The control messages specify the new values for the control parameters, as well as the resource conditions under which these new settings are valid (because of user preference constraints). Upon receiving them, the steering agent sets up the new configuration, which can cause the execution of the application specific transitional functions, in preparation for the activation of the corresponding configuration.

Example. Figure 9 shows how the above framework facilitates runtime adaptation in the Active Visualization application. The four figures show how the application reacts to (a) a change in network bandwidth from 500 KBps to 50 KBps after 25sec into the experiment (the framework responds by switching compression methods); (b) a change in CPU share from 90% to 40% after 30sec into the experiment (the framework responds by degrading image resolution); and (c), changing CPU share, as described above, but with the additional requirement that the average transmission time of user interactions be kept below 1sec (the framework responds by reducing the fovea size). Note that the response time is kept below 1sec, as shown in Figure 9 (c), while the transmission time increases, corresponding to the larger fovea size. Each of the plots shows the application behavior with adaptation (bold line) as well as with the configurations before and after adaptation.

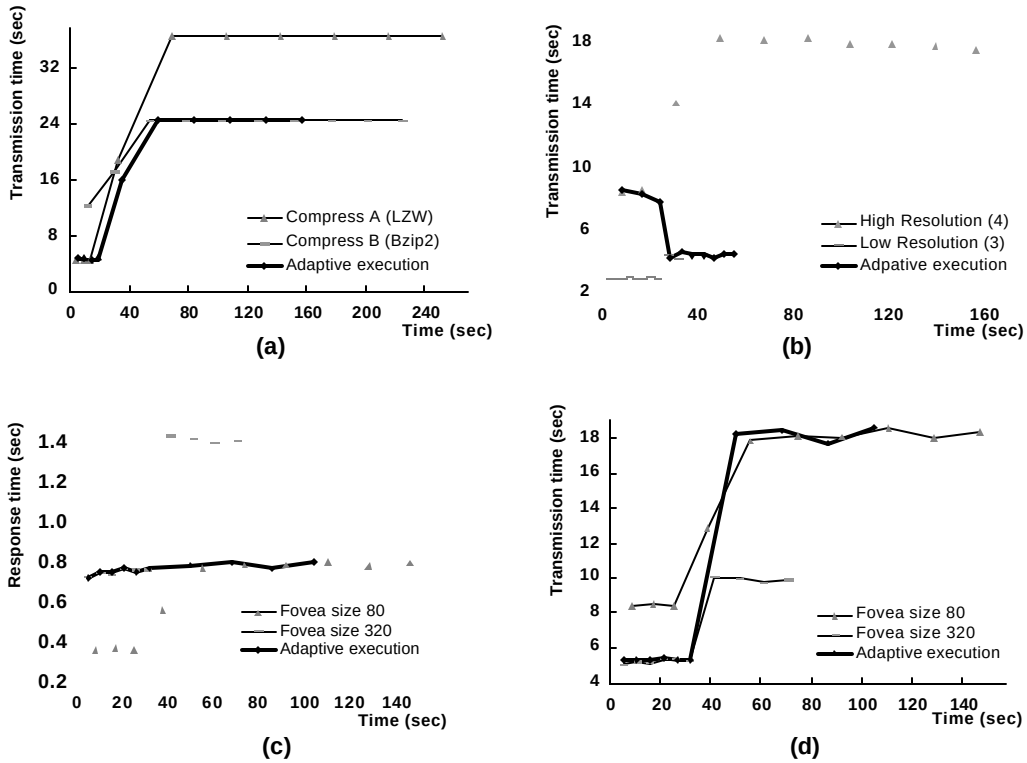


Figure 9: Runtime Adaptation in the Active Visualization Application.

6 CANS: Composable Adaptive Network Services Infrastructure

CANS is an application-level infrastructure for injecting application-specific components into the network that investigate application-independent mechanisms for adaptation. The CANS view of adaptation is one where typed interactions between legacy application components are customized to their execution environment (note that component behaviors are indirectly controlled by customizing the interactions) by the automatic insertion of application-specific intermediary components.

The CANS research focuses on three challenges: (a) efficient and dynamic composition of individual components; (b) distributed adaptation of injected components in response to system conditions; and (c) support for legacy applications and services. CANS addresses these challenges by constructing networks that include *applications,* stateful *services,* and *data paths* between them built up from mobile soft state objects, called *drivers.* Drivers implement a standard interface, permitting efficient composition and semantics preserving adaptation. Both services and data paths can be dynamically created and reconfigured: a planning and event propagation facility enables distributed adaptation, while a flexible type based composition model dictates how new services and drivers are going to be integrated with the existing ones. CANS provides three adaptation modes to permit cost functionality tradeoffs: intra-component, by reconfiguring data

paths, and by creating new services and data paths. Legacy components plug into CANS using delegation and an interception layer that transparently virtualizes network bindings, currently known as TCP sockets.

6.1 CANS Architecture:

CANS views networks as consisting of *applications*, *services*, and *data paths* that connect them (see Figure 10). CANS extends the notion of a data path, traditionally limited to data transmission between end points, to include application specific components that are dynamically injected by end services, applications, or other entities; these components adapt the data path to physical link characteristics of the underlying network and properties of end devices.

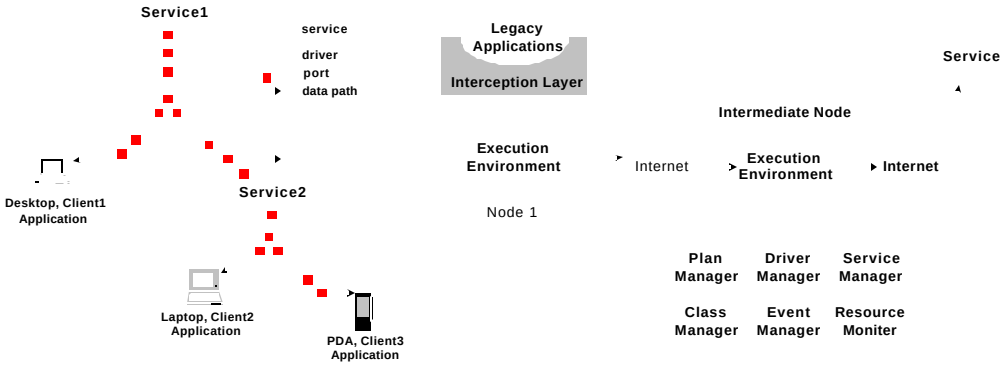


Figure 10: CANS logical architecture (left) and physical realization using per-node execution environments.

Components are self-contained pieces of code that can perform a particular activity, e. g., protocol conversion, or data transcoding. Components operate on *typed* data streams and are connected with each other, based upon the compatibility of the output and input types. Injected components come in two flavors: stateful *services* and mobile soft state objects, called *drivers*. Services extend the original data path to multiple hops, and drivers generalize the traditional notion of a data path to include data transformation, in addition to transmission. The primary reason for distinguishing between drivers and services is to ensure efficiency.

CANS data paths are created dynamically, using information about user preferences, properties of services and client applications, as well as characteristics of the underlying platform. The components, which constitute a data path, the interconnections amongst them, and their internal configuration parameters can all be modified at run time. Modifications are triggered based on either system events (e. g., breaking of a network link) or component initiated events. The CANS infrastructure provides support to efficiently reconfigure data paths, while preserving application semantics.

The CANS network is realized by partitioning the services and data paths onto physical hosts, connected using existing communication mechanisms. The CANS Execution Environment (EE) serves as the basic runtime environment on these hosts, and includes the following functional modules: *class manager, plan manager, driver and service manager, event manager,* and *resource monitor.*

The class manager handles the downloading of component code and the instantiation of the components. The plan manager is responsible both for creating the initial plan comprising drivers, services, and data paths, in response to a request trapped by the interception layer. The plan manager is also responsible for replanning, in response to system conditions. The driver and service manager maintain information about deployed drivers and manage data path operations, including inserting new drivers, creating new services, and reconfiguring existing paths, as required. The event manager is responsible for receiving both system level and component level events, and propagating these on to interested components. The resource monitor monitors the system conditions, such as the CPU availability or network interface state, informing the event manager when registered trigger conditions are fired.

6.2 CANS Components

CANS components include drivers, services, and auxiliary components that interconnect execution environments, applications, and legacy services.

Drivers: Drivers serve as the basic building block for constructing adaptation capable, customized data paths. Drivers are standalone mobile code modules that perform some operation on the data stream. However, to permit their efficient composition and dynamic low overhead reconfiguration of data paths, drivers are required to adhere to a restricted interface. Specifically,

1. Drivers consume and produce data using a standard *data port* interface, called a `DPort`. `DPorts` are typed (more details are provided below) and distinguished, based on whether they are being used for input or output.
2. Drivers are *passive,* moving data from input ports to output ports in a purely demand-driven fashion. Driver activity is triggered only when one of its output ports is checked for data, or one of its input ports receives data.
3. Drivers consume and produce data at the granularity of an integral number of application-specific units, called *semantic segments.* These segments are naturally defined, based on the application, e. g., an HTML page or an MPEG frame. Informally, this requirement ensures that the data in an input semantic segment that can only influence data in a fixed number of output segments, thus permitting the construction of data path reconfigurations and error recovery strategies that rely upon retransmission at the granularity of semantic segments. Note that this property only refers to the logical view of the driver, and admits physical realizations that transmit data at any convenient granularity, as long as segment boundaries are somehow demarcated (e. g., with marker messages).
4. Drivers contain only *soft state,* which can be reconstructed simply by restarting the driver. Stated differently, given a semantically equivalent sequence of input

segments, a softstate driver always produces a semantically equivalent sequence of output segments. For example, a Zip driver that produces compressed data will produce semantically equivalent output (i. e., uncompressed to the same string) if presented with the same input strings, MPEG 500x200.

The first two properties facilitate the dynamic composition and efficient transfer of data segments between multiple drivers that are mapped to the same physical host (e. g., via shared memory). Moreover, they permit driver execution to be orchestrated for optimal performance. For example, a single thread can be employed to execute, in turn, multiple driver operations on a single data segment. This achieves nearly the same efficiency; the module indirect function calls overheads, as if the driver operations were statically combined into a single procedure call.

The semantic segments and softstate properties enable low-overhead dynamic adaptation, either within a single driver or across data path segments while preserving application semantics. The driver interface permits a driver to create and listen to events, thus facilitating its participation in distributed adaptation activities.

Type-based Composition: The composability of CANS components (both drivers and services) is determined by the compatibility of the type information associated with the input and output ports being connected. The types used in CANS integrate two closely related concepts: *data types* and *stream types* (see Figure 11).

CANS data types are the basic unit of type information, represented by a type object that in addition to a unique type name can contain arbitrary attributes and operations for checking type compatibility. Traditional mechanisms, such as type hierarchies, can still be used to organize data types; however, our scheme permits flexible type compatibility relationships that can not be expressed by simply matching the type names. For instance, it is possible to define a CANS type for MPEG data, which contains attributes for defining the frame size. An `MPEG` type can be considered to be compatible with another `MPEG` type, as long as the former's frame size is smaller than the latter's, naturally capturing the behavior that a lower resolution MPEG stream can be played on a client platform capable of displaying a higher resolution stream.

CANS stream types capture the aggregate effect of multiple CANS drivers operating upon a typed data stream. Stream types are constructed at run time, and are represented as *stacks* of data types. Operations allowed on stream types, include *push, pop, peek,* and *clone,* have the standard meanings.

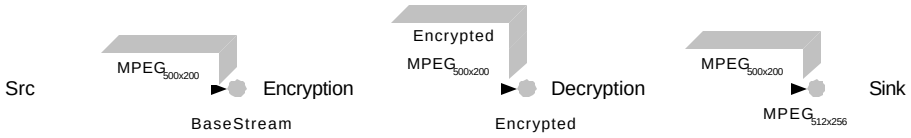


Figure 11: CANS data and stream types.
Stream types retain run-time representations of operations on basic data types, to permit appropriate composition.

Services: The second core CANS component are *services.* Unlike drivers that represent rigidly constrained, mobile, and softstate adaptation functionalities, services can export data by using any standard internet protocol (e. g., TCP or HTTP) that encapsulates more heavyweight functions, process concurrent requests, and maintain a persistent state. The different interface requirements of drivers and services stem from the observation that the most current services, distributed in the internet, are legacy in nature: their source code is generally unavailable, while rewriting or modifying them is impractical. The price paid for not adhering to a standard interface is that unlike driver migration, CANS does not explicitly support service migration; a service individually determines how it manages its own state transfer. This design choice reflects the view that services are migrated infrequently and doing so requires protocols that are difficult to abstract.

CANS provides applications with a general platform to create, compose, and control services across the network. A service is required to register itself, thus identifying the data types it supports, and optionally providing a *delegate object* that can control the service and act on its behalf in interactions with the rest of the CANS. The delegate object implements a standard interface that consists of activated and suspended services, and received CANS events. Service composition is similar to driver composition, in that it uses types supplied at registration time.

Communication Adapters: Communication adapters are auxiliary CANS components, which transmit data *physically* across the network to connect drivers that span different nodes. To achieve this, these components expose the same `DPort` appears to other drivers just as a regular communication. Adapters also support two additional logical connections: (1) between applications and drivers; and (2) between a driver and a service that export data using an interface other than `DPort.`

To provide the above functionality, adapters establish physical communication links between application (see below) and execution environments; between two execution environments; and between an execution environment and a service. Multiple logical connections can be multiplexed on this single physical link; the latter can exploit transport mechanisms, best matched to the characteristics of the underlying network. Communication adapters can additionally encapsulate behaviors that permit them to adapt to and recover from minor variations in network characteristics. For instance, these adapters can be written to use one of several network alternatives, automatically transitioning between them to improve performance. The continuity semantics upon such reconnection are dictated by the requirements of the data types associated with the adapter's ports.

Support for Legacy Applications: The CANS infrastructure supports both CANS-aware and CANS-oblivious applications. The former just hook into the driver and service interfaces, described earlier. The latter require more support but can be easily integrated because of our focus on stream based transformations on the data path. Our solution relies on an *interception layer* that is transparently inserted into the application and is virtualized through its existing network bindings. The interception layer is injected using the API interception and has the structure shown in Figure 12.

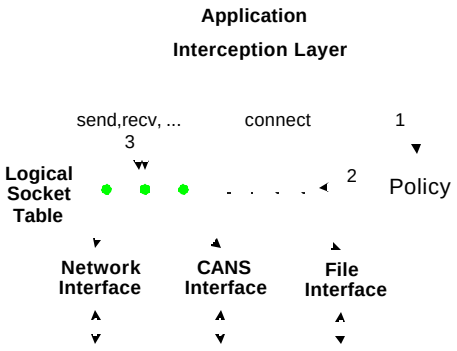


Figure 12: CANS supports legacy applications by relying on an interception layer that reinterprets network calls.

6.3 Distributed Adaptation in CANS

CANS supports three modes of adaptation in response to dynamic changes in system characteristics: (1) *intra-component adaptation,* where each service or driver detects and adapts to minor resource variations on its own; (2) *data path reconfiguration and error recovery,* where the data path undergoes localized changes involving insertion, deletion, and reordering of drivers; and (3) *replanning,* where the existing data paths are torn down and new ones are constructed to respond to the large-scale system variations. These three modes represent different points on the cost functionality spectrum, enabling the system to respond to system events with the least overhead possible. To the best of our knowledge, CANS is unique in providing system support for data path reconfiguration.

Each CANS driver and service can incorporate its own adaptation behavior that may or may not be coordinated with the adaptation in other components. For example, a frame dropping component can alter its policies upon detecting different levels of backpressure on its output buffers. Note that adaptation in a single component is completely isolated as long as its effect is restricted within a single semantic segment.

To trigger adaptation, CANS provides distributed event propagation support, permitting components (including delegate objects for legacy services) to raise arbitrary events as well as listen for specific ones. Event support is provided by a per execution environment *Event Manager***,** which is responsible for catching, firing, and transmitting events across the network. Event raising and firing are implemented using simple method calls and callback functions associated with the relevant component.

There are two major types of CANS events: events from the local resource monitor, indicating a change in resource status, and events from components on the data path. The first kind of events is sent only to local components that register themselves as interested listeners. The second kind of events, issued by components along a data path, are first sent to the *plan event delegate*, which is responsible for propagating the event along the data path as well as handling plan specific events, such as events to trigger replanning.

Insertion, deletion, or reordering of drivers along an active data path provides great flexibility in responding to a range of resource variations and link/ node failure. However, a fundamental problem is that any such reconfiguration must preserve application semantics. CANS mechanisms for reconfiguration focus on maintaining semantic continuity and exactly-once semantics. Specifically, any scheme must take into account the fact that the portion of the data path affected by the reconfiguration can have stream data that has been partially processed: in the internal state of drivers, in transit between execution environments, or data that has been lost due to failures. Note that although the soft-state requirement permits us to restart a driver, it does not provide any guarantees on semantic loss or in order reception.

CANS formulates the driver selection problem as one of finding a type-compatible sequence of components transforming the type provided by the source component into one required by the destination, and which additionally optimize the overall quality of the path. As part of the CANS research, we have developed a dynamic programming algorithm that does not only solve this problem efficiently, but it also enables the independent reconfiguration of sub-portions of the data path.

6.4 CANS Prototype and Performance Benefits

A CANS prototype has been built using the Java programming language, and incorporates the architecture and algorithms described above. Its functionality has been verified using several applications, including a web access scenario that involves mobile devices and weak network connections, as well as an image streaming application that tests the effectiveness of continuous adaptation.

Figure 13 shows the normalized execution time seen by users in the web access scenario, who attempt to download a series of web pages using the various web clients and network connectivity options listed on the x-axis. To take an example, PocketPC-19.2Kbps-Pentium III-10Mbps, refers to a configuration where a mobile client attempts to browse the web using a Pocket PC device with cellular network connectivity, and where the gateway to the wired network is a Pentium III –class desktop with 10 Mbps Ethernet-like connectivity. The different bars correspond to the execution times seen by different CANS paths (Direct refers to the situation where there are no intermediate components). Figure 13 also shows the structure of each of these paths, which are made up of **M**ultiplexer and **D**emultiplexer components at either end, text **Z**ip/**U**nzip, and Image **R**esizer/**F**ilter components to reduce the bandwidth requirements of the downloads. The paths that are automatically created and installed by the CANS framework for each client and network configuration are identified on the x-axis. The figure shows that not only do automatically generated CANS paths outperform other paths, but that additionally a different dynamic path is in fact required for different network configurations. The latter result validates the benefits of a CANS-like dynamic approach.

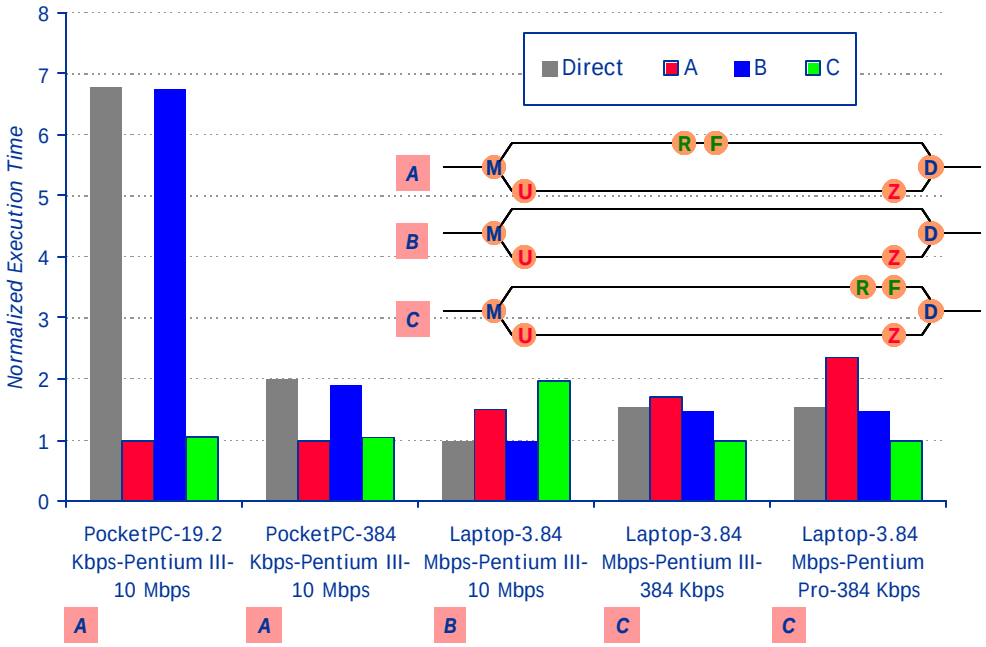


Figure 13: Normalized execution time seen by clients accessing web pages for CANS paths created under diverse network conditions.

Figure 14 shows the effectiveness of CANS vis-à-vis continuous dynamic adaptation, where both the quality of the generated data path and the overheads of adaptation must be small enough for the end result to be acceptable. The figure shows the throughput (in frames/second) seen by an image streaming application running on a client machine as the bandwidth of the network connection between the client and the server machine is gradually reduced from 150 KBps to 10 KBps, and then increased back up to 150 KBps over a 20 minute period. The application requirement is for a throughput of 8-15 frames/second, which is delivered by CANS by continually adapting to the changes in network characteristics. The adaptations insert and optionally modify the parameters of the Image Resizer and Filter components discussed earlier. The different adaptations that need to be performed in this experiment correspond to the plateaus shown in the quality curve in Figure 14.

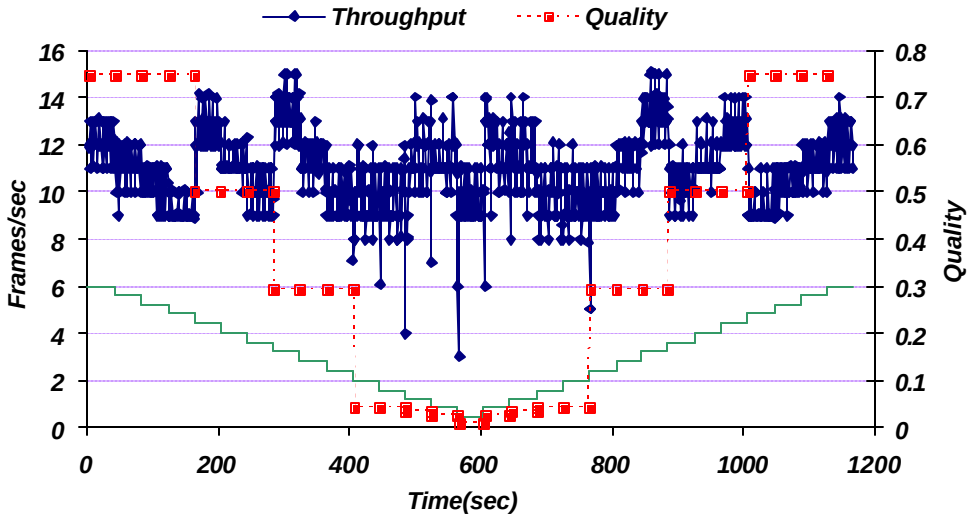


Figure 14: Throughput and Quality seen by an image streaming application running on top of the CANS framework as network bandwidth is continually lowered(green curve) from 150 KBps down to 10 KBps and then increased.

7 Distributed Resource Sharing Agreements

The resources in dynamically formed computing communities are shared among the members of the community. Although current uses of such sharing, motivated by work on computational grids, focus on distributed systems built from components that belong to either a single or a small number of administrative domains (e. g., US supercomputer sites), the growth of the internet has resulted in newer models of resource sharing that span across multiple domains. For example, an increasing number of companies find themselves having to share information in their databases with others. Companies with limited budgets may want to share hardware (e. g., network bandwidth and disk capacity) with others. The growing popularity of application specific providers (ASPs) exemplifies a situation where the ASP is sharing its resources with several client organizations.

In each of the above scenarios, the resources are owned by different organizations and span across multiple administrative domains. Consequently, resource sharing is governed by *sharing agreements* between participating entities. For example, organization *A* and *B* have resource sharing agreements between them so that *A* can use 30% of *B's* network bandwidth, and in return, *B* can use 20% of the CPU power of *A's* supercomputer. These agreements specify both the obligations and privileges of the participants, while the executions of applications should follow or be forced to follow these agreements. Moreover, agreements must be enforced in the presence of heterogeneous resource types, thus dynamically changing the user set and resource availability. Also, sharing agreements can be transitive, i. e., one principal can share resources from another via chain of sharing agreements even though the two principals do not have any direct

sharing agreements. Unfortunately, most existing resource management infrastructures such as Condor, Globus, and Legion provide very limited support for expressing and efficiently enforcing such sharing agreements. This support typically takes the form of "matching" up requests with compatible resource types and does not factor in any capacity restrictions implied by the agreements. The enforcement of capacity constraints is usually the responsibility of the end points in those systems.

As part of our work on this project, we developed a novel approach for expressing resource-sharing agreements, which extends the concepts of *tickets* and *currencies,* originally proposed in the context of uniprocessor operating system scheduling. Our approach abstracts out resource heterogeneity by uniformly representing resource capacities in terms of tickets with differing values. Thus, diverse resources such as an Intel Pentium II CPU and an Intel Pentium III CPU can both be represented as a CPU ticket with different value. Agreements themselves are captured in terms of other kinds of tickets that are issued by currencies. The value of a currency fluctuates dynamically as a function of changing resource availability and affects the actual value of the tickets. This allows capturing of both absolute and relative sharing agreements.

The above expression scheme is complemented by a global resource allocation algorithm that uses linear programming techniques to enforce these agreements, thus automatically factoring the transitive availability of resources via chained agreements. When multiple scheduling choices are available, the algorithm chooses the one that satisfies a desired global objective, such as fairly allocating system resources or alternately maximizing service provider revenues. By exposing resource-sharing agreements to the resource scheduler, the scheduler is able to make more informed decisions that take both the resource availability and resource sharing agreements into account.

The expression scheme and enforcement architecture have been implemented and evaluated in the context of an ASP-level sharing architecture, where distributed clients attempt to access hosted network-accessible services (governed by agreements between the owning organization and the ASP) via distributed network of redirector nodes. The redirector nodes perform admission control and route client requests to appropriate servers, as necessitated by the sharing agreements. Experiments show that the infrastructure is not only capable of capturing a range of agreements but also efficiently enforcing desired properties despite fluctuations in client request traffic, and server and redirector node unavailability.

7.1 Expressing Sharing Agreements

A resource management system requires precise expression of three kinds of information: resource availability, resource requests, and agreements between users. Resources refer to both physical entities such as CPU and disk, as well as to logical entities such as access rights. Both resource availability and resource requests are represented as vectors, with entries quantifying the quantity or need for each different kind of resource. Our approach expresses actual resource capacities and agreements between participants using a uniform framework that permits the convenient computation of dynamic resource availability for each individual participant.

A Taxonomy of Agreements Sharing agreements between the owner of a resource and its users define both which users can access the resource and any capacity constraints governing such access. For example, owner *A* might specify that its resources are available for use by users *B* and *C* but not by user *D*. Moreover, *A* might enable *B* to use a larger fraction of the resource than *C*. Agreements can be either *granting* or *sharing* in nature, according to who can use the resources after the agreement is applied. The former refers to the case where the grantor gives up the resource to the grantee, i. e., the grantor cannot use the resource itself after granting unless it revokes the resource from the grantee (agreement ends). With sharing agreements, both the grantor and grantee have the rights to use the resource. In this paper, we focus on consumable resources, such as CPU and disk bandwidth, which can only be used by one principal at a time. Another dimension that can be classified along which agreements is the expression of the quantity constraints on sharing agreements. Agreements can be either *absolute* or *relative*. In absolute agreements, the quantity is a constant (e. g. 10MB disk space), while in relative agreements, the quantity varies as a function of available resources (e. g. 50% of the available CPU). Relative agreements can be used to ensure that a certain fraction of a principal's resources is always available for its own use, even when the available resources decrease due to intensive usage.

Tickets and Currencies The primary challenge with expressing sharing agreements is to capture the heterogeneous nature and dynamically changing availability of resources that use an agreement to determine which resources are accessible and in what quantity. We use the concepts of *tickets* and *currencies,* originally proposed in the context of a randomized uniprocessor scheduling mechanism, called Lottery Scheduling, to address this challenge.

Tickets are abstract entities that differ in type and value. In our approach, tickets are used to encapsulate both access and capacity constraints governing resource usage. Possessing the right ticket type permits access to the resource while the ticket value determines the resource quantity that can be accessed. Tickets can be either *absolute* or *relative*. An absolute ticket has the right to use an absolute quantity of resource (e. g., 10MB of disk), while the value of a relative ticket depends on the *issuing* currency.

Currencies denominate tickets. Each currency is backed (or funded) by tickets and in turn issues its own tickets. The agreements between *A* and *B* can be represented by *A's* currency by issuing a ticket to support *B's* currency, which means that *B* can use part of A's resources (see Figure 15). The value of an absolute ticket is its face value. And a the relative ticket's real value is computed by multiplying the value of the currency from which it is issued by its share of all the amount issued by that currency. The value of a currency is determined by the summation of all the backing tickets (both absolute ones and relative ones).

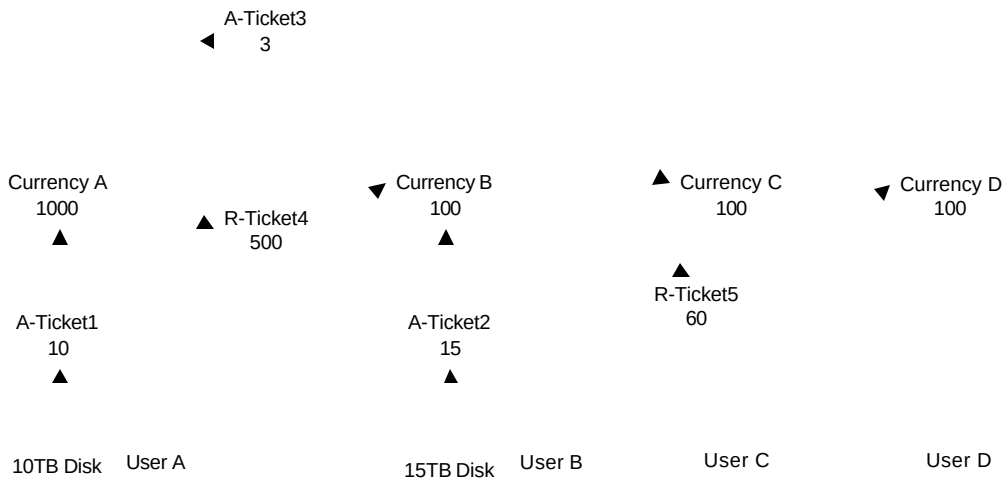


Figure 15: Use of Tickets and Currencies to express Sharing Agreements.

Figure 15 shows an example of the use of tickets and currencies for expressing sharing agreements. This system has four principals: A, B, C, and D, and two disk resources of 10 TB and 15 TB capacity that are owned by the principals *A* and *B,* respectively. These resources are represented by two absolute tickets (ATicket1 and ATicket2), which fund currencies *A* and *B,* respectively. Principal *A* has an absolute agreement with principal *C* to share 3 TB of its resources, and a relative agreement with principal *B* to share 50% of its available resources. The former is captured by an absolute ticket (ATicket3) with value 3 and the latter by a relative ticket (RTicket4) with a face value of 500 that is issued by currency *A,* which has a face value of 1000. Thus, the real value of RTicket4 is 10×500/1000 = 5. This relative ticket boosts the value of currency *B* to 5+15 =20, increasing the amount of disk resources available to jobs submitted by principal *B.* Principal *B,* in turn, has a relative agreement with principal *D* that is captured via relative ticket RTicket5 of face value 60, which is issued by currency *B* and has a face value of 100. Note that the true value of this ticket is 20×60/100 = 12, which implicitly integrates the resources available to principal *D* via its direct agreement with *B,* as well as its transitive agreement with *A* (via *B).*

Thus, the actual resource capacities in a system are expressed using absolute tickets that fund the owner's currency and agreements between participants, and take the form of absolute or relative tickets issued by one and backing the other's currency. The quantity constraint of this sharing agreement is the value of the ticket. Notice that the real value of relative tickets can change dynamically as more supporting tickets join the issuing currency or as some supporting tickets leave as a consequence of changing resource sharing agreements. In addition, we can inflate or deflate the currencies by increasing or decreasing the number of units in the currency, similar to the inflation caused by the government printing more paper money. Furthermore, new currencies can be created in addition to the default per participant currencies: these permit a participant to decouple

the quantity of resources transferred along some subset of its agreements from fluctuations in another subset.

Although we have restricted our discussion here to the case where a single ticket type represents one resource, this mechanism can be extended to handle multiple views of the same resources by enabling resources backing multiple ticket types. This is useful in several situations; for example, the disk bandwidth resource can be viewed as two kinds of resources: read bandwidth and write bandwidth.

The basic mechanism described above has also been extended to capture more general agreement structures. Specifically, these extended agreements capture the access privileges that principal j has on principal i's resources over a time window, and are modeled as a tuple: $[lb_{ij}, ub_{ij}]$ representing the lower bound (guaranteed reservation during overload) and an upper bound. The lower bound is different from other reservation systems in which the reserved resources are put aside waiting for requests. In our model, the resources reserved for principal j can be used by others if j does not use them. This ensures better resource utilization. In addition, agreements are interpreted dynamically: changes in a principal's resource levels affect the amount available to others via agreements. Extending ownership to include resources that a principal obtains via its own agreements enables transitive flow of resources. There are two types of tickets that represent the [lb, ub] form of agreements: mandatory and optional, and two corresponding values for each currency. A mandatory ticket corresponds to the lower bound (lb_{ij}) of an agreement, and an optional ticket represents the difference between the upper and lower bound (ub_{ij} - lb_{ij}).

Despite the generality afforded by our expression mechanism, in practice, we expect most sharing agreements to fall into one of the following structures:

Complete: Each participant has sharing agreements with every other participant. This situation is most likely to occur when there is a small number of participants.

Sparse: Every participant has sharing agreements only with a relatively small number of other participants. This structure happens in situations where the number of participants is relatively large.

Hierarchical: In this structure, users are divided into groups. Inside a group, users have complete resource sharing. Between groups there are higher level sparse sharing agreements.

7.2 Enforcing Sharing Agreements

Coordinated support is needed for enforcing sharing agreements that involve distributed requestors and resources. An additional consideration is that the scheduler must keep track of resource availability via both direct and transitive agreements. Figure 16 (left) shows our agreement enforcement architecture, which coordinates multiple redirector nodes to schedule requests from distributed clients to servers. The ticket/currency-based scheme enables redirector functionality to be oblivious to the specific resource type being accessed. Each redirector node logically maintains a set of queues (see Figure 16 (right)), one for each principal, and forwards a subset of these requests to servers every time the

window satisfies the mandatory (MC_i) and optional (OC_i) request processing rates the principal is entitled to, computed as described below. Stated differently, the redirector nodes perform admission control at the system edges to forward incoming request traffic to server resources so that the desired service levels are obtained. We first describe the queuing algorithms assuming a single redirector, and then extend these to multiple redirectors distributed across a wide-area network.

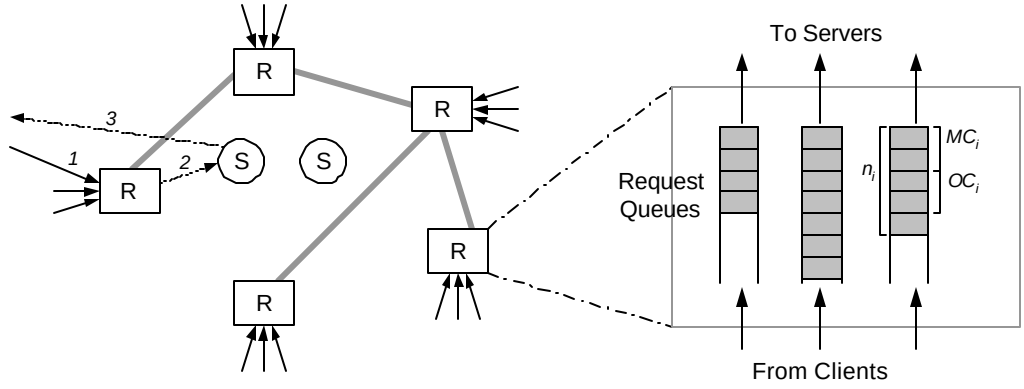
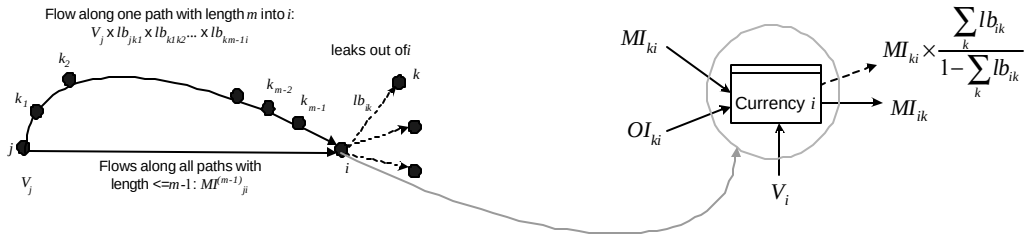


**Figure 16: Coordinated redirector nodes (R)
queue and forward requests to server resources (S)**

Scheduling using a single redirector: Given an agreement graph as input, the scheduler must: (1) determine per-principal mandatory and optional request processing rates implied by using both direct and indirect agreements; and (2) based upon the request patterns observed at run-time, decide how to schedule each principal's requests to optimize a global system-wide criterion.

For quasi-static agreement structures, mandatory and optional request processing rates can be pre-computed based on the agreement graph. For simplicity, our description assumes a single resource type: the agreement matrix consists of elements representing direct agreements. Principal *i*'s physical resources are represented by their aggregate capacity V_i, scaled in terms of the average requirements of a request. To compute the mandatory (MC_i) and optional (OC_i) processing rates for requests of a particular principal i, we first calculate the *flow* of mandatory and optional resources from each principal *j* to *i* (see Figure 17(a)). The flow of resources can be expressed in terms of a recurrence relation that involves the number of direct agreements along with the path. Formula 1 and 2 in Figure 17 give the expressions for these flows. Formula 1 shows how the mandatory resources flow along the mandatory tickets (lower bound) from one currency to another: the additional factor excludes the mandatory value *i* passes along to others (i.e. which leaks out of *i*), parthas shown in figure 1. Formula 2 is more complex, since it illustrates how the mandatory currency values contribute to the mandatory resources (via mandatory tickets) up to a particular point in the path, and thereafter to optional resources (via optional ticket at the specific point, and via agreement upper bounds beyond that point). The summation constraints ensure that there is no cycle along the transitive agreement path.



$$MI_{ji}^{(m)} = MI_{ji}^{(m-1)} + V_j \times \sum_{\substack{1 \le k_1, \Lambda, k_{m-1} \le n \\ k_p \ne i, j, k_q (\forall p \ne q)}} lb_{jk_1} lb_{k_1 k_2} \Lambda \; lb_{k_{m-1} i} (1 - \sum_k lb_{ik}) \quad (1)$$

$$OI_{ji}^{(m)} = OI_{ji}^{(m-1)} + V_j \times \sum_{\substack{1 \le k_1, \Lambda, k_{m-1} \le n \\ k_p \ne i, j, k_q (\forall p \ne q)}} \sum_{r=1}^{m-1} lb_{jk_1} lb_{k_1 k_2} \Lambda \; lb_{k_{r-1} k_r} (ub_{k_r k_{r+1}} - lb_{k_r k_{r+1}}) ub_{k_{r+1} k_{r+2}} \Lambda \; ub_{k_{m-1} i} \quad (2)$$

$$MC_i = V_i + \sum_{k \ne i} MI_{ki} - \sum_{k \ne i} MI_{ik} \quad (3)$$

$$OC_i = \sum_{k \ne i} OI_{ki} + \sum_{k \ne i} MI_{ik} + \sum_{k \ne i} MI_{ki} \times \frac{\sum_k lb_{ik}}{1 - \sum_k lb_{ik}} \quad (4)$$

Figure 17: Computation of mandatory and optional request processing rates.

After the algorithm learns the MC_i and OC_i for each principal i, it must determine in each time window, what subset of the n_i requests in the i's queue (Figure 17(right)) must be forwarded to the servers. This decision should respect sharing agreements and at the same time optimize a global metric. An example is a system where the objective is to minimize the maximum response time over all the client requests. This optimization can be formulated as a linear-programming problem that involves constraints for satisfying the agreements and the access level formulae 3 and 4, computed earlier. By solving this linear programming model, the redirector obtains the queuing schedule that achieves the desired objective.

Coordinated Scheduling across Redirectors: The queuing strategy, discussed above, is not very scalable, because it assumes that there is a single redirector that sees all the requests. Our general solution extends the queuing algorithm to a distributed setting by observing that the single-node solutions can work on local redirector queues in the same way as before, as long as decisions about the fraction of the local queue to transmit to servers are based upon global values of the total queue lengths for each principal. Note that the queue length is an aggregate quantity of the system and can be computed much more efficiently than a neighbor-wise exchange of queue statistics. In particular, we organize the multiple redirector nodes into a *dynamic combining tree* network. Redirectors located at the leaves of this tree periodically send queue length information to their parents. An intermediate node in the tree waits for information from its children, adds its local queue information to this, and passes on the information to its parent. When the queue length information reaches the root, this node sends the final aggregate information down the tree, effectively using the combining tree as a broadcast tree.

7.3 Prototype and Performance Evaluation

The resource sharing architecture described above has been implemented both at Layer 4 (as an architecture that intercepts, buffers, and redirects IP packets) and at Layer 7 (as an HTTP proxy) of the ISO/OSI protocol stack. The Layer 4 implementation builds upon the two open-source packages developed byother researchers, the Linux Virtual Server infrastructure and the Spread group communication package. Figure 18 shows the overall structure of this implementation, depicting the packet flow and interaction with a linear equation solver.

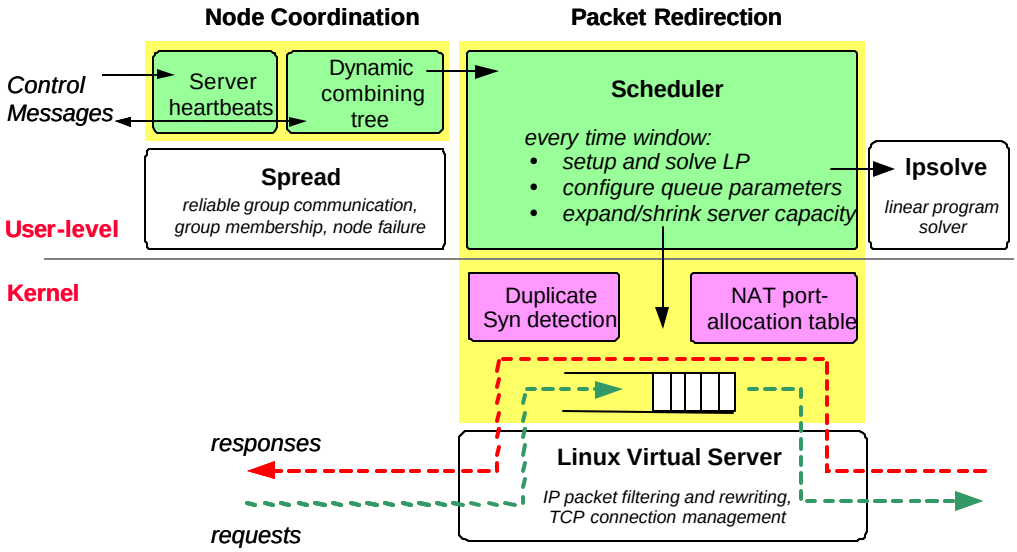


Figure 18: Architecture of the Layer 4 implementation of the sharing agreement enforcement architecture.

The functionality and performance of the prototypes have been evaluated using synthetic workloads generated from the WebBench web server evaluation infrastructure. Clients run the httperf program by sending requests to Apache web servers through a redirector network. Figure 19 shows two sample performance plots involving different numbers of client, server, and redirector nodes. Figure 19 (top) verifies that the resource sharing architecture is capable of respecting sharing agreements (despite server and redirector faults, which are seamlessly handled by the group communication infrastructure). Figure 19 (bottom) shows that the infrastructure additionally optimizes a global criterion (provider income in this case), despite distributed decision making.

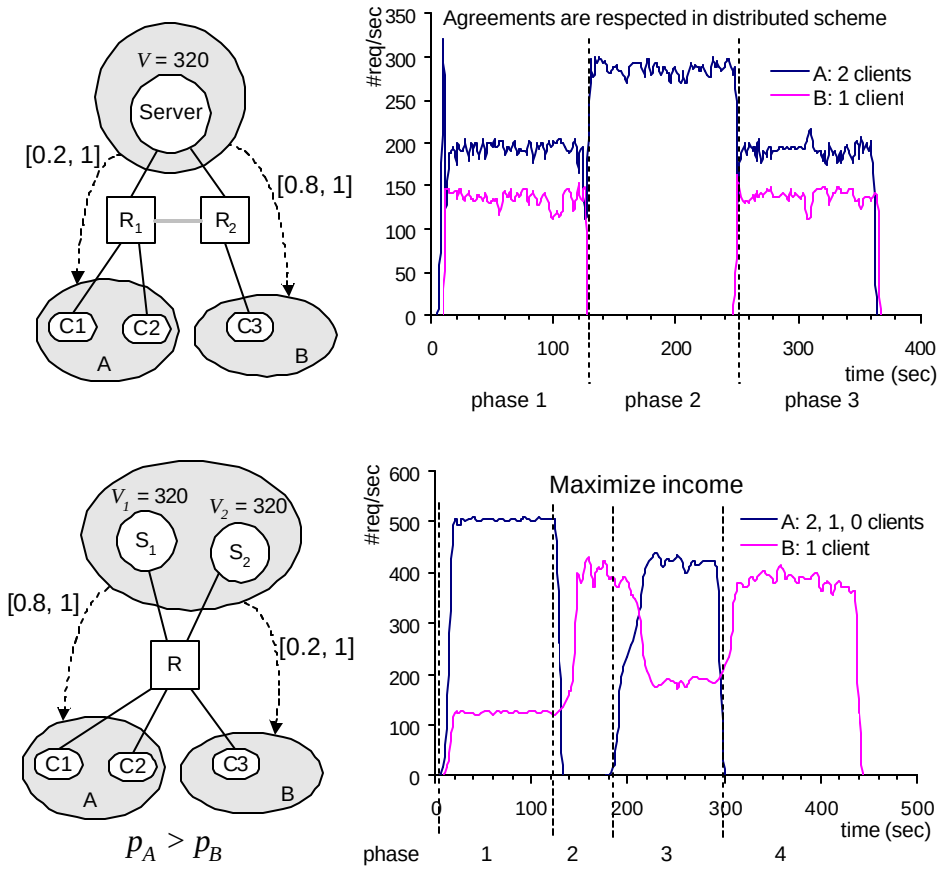


Figure 19: Evaluation of the resource sharing agreement architecture showing that agreements are respected (top) and that a global objective (provider revenue in this case) is maximized (bottom).

8 Conclusions

The research work was conducted by the computing communities projects, and it addressed the issues of managing the quality of service of large distributed systems through the use of innovative techniques, such as virtualization, adaptation, and tunability.

9 Publications

X. Fu and V. Karamcheti, Automatic Creation and Reconfiguration of Network-Aware Service Access Paths, *NYU Computer Science TR 2001-824*, revised December 2002.

T. Zhao and V. Karamcheti, Enforcing Resource Sharing Agreements among Distributed Server Clusters, *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.

Tom Boyd and Partha Dasgupta, Process Migration: A Generalized Approach Using a Virtualizing Operating System, 22nd International Conference on Distributed Computing Systems (ICDCS-2002), Vienna, July 2002,

Tom Boyd and Partha Dasgupta, Preemptive Module Replacement Using the Virtualizing Operating System, Workshop on Self-Healing, Adaptive and Self-Managed Systems (SHAMAN), New York, June 2002.

Shu Zhang, Mujtaba Khambatti and Partha Dasgupta, Process Migration through Virtualization in a Computing Community, 13th IASTED International Conference on Parallel and Distributed Computing Systems (PDCS2001), Anaheim, CA, August 2001.

F. Chang and V. Karamcheti, A Framework for Automatic Adaptation of Tunable Distributed Applications, *Cluster Computing: The Journal of Networks, Software and Applications*, Volume 4, Number 1, 2001.

F. Chang, Automatic Adaption of Tunable Distributed Applications, *Ph.D. Thesis, Department of Computer Science, New York University*, 2001.

X. Fu, W. Shi, A. Akkerman, and V. Karamcheti, CANS: Composable, Adaptive Network Services Infrastructure, *USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

X. Fu, W. Shi, and V. Karamcheti, Automatic Deployment of Transcoding Components for Ubiquitous Network-Aware Access to Internet Services, *NYU Computer Science Technical Report CS-TR-2001-814*, March 2001.

F. Chang, V. Karamcheti, and Z. Kedem, Exploiting Application Tunability for Efficient, Predictable Resource Management in Parallel and Distributed Systems, *Journal of Parallel and Distributed Computing*, Vol. 60, pp. 1420-1445, 2000.

F. Chang, A. Itzkovitz, and V. Karamcheti, User-level Resource-Constrained Sandboxing, *4th USENIX Windows Systems Symposium*, August 2000.

T. Zhao and V. Karamcheti, Expressing and Enforcing Distributed Resource Sharing Agreements, *SC2000: High Performance Networking and Computing Conference*, November 2000.

F. Chang and V. Karamcheti, Automatic Configuration and Run-time Adaptation of Distributed Applications, *Ninth IEEE Symposium on High Performance Distributed Computing*, August 2000.

Tom Boyd and Partha Dasgupta, Virtualizing Operating Systems for Seamless Distribution, 12th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2000), November 2000.

R. Nasika and P. Dasgupta, Transparent Migration of Distributed Communicating Processes, 13th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS-2000), August 2000.

T. Boyd and P. Dasgupta, Injecting Distributed Capabilities into Legacy Applications Through Cloning and Virtualization. The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), July 2000.

X. Fu, H. Wang, and V. Karamcheti, Transparent Network Connectivity in Dynamic Cluster Environments, *4th Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'00)*, January 2000.

P. Dasgupta, A. Itzkovitz, and V. Karamcheti, Active Files: A Mechanism for Integrating Legacy Applications into Distributed Systems, *International Conference on Distributed Computing Systems (ICDCS)*, April 2000.

A. Baratloo, P. Dasgupta, V. Karamcheti, and Z. Kedem, Metacomputing with MILAN, *Heterogeneous Computing Workshop*, *International Parallel Processing Symposium (IPPS'99)*, April 1999.

P. Dasgupta, V. Karamcheti, and Z. Kedem, Transparent Distribution Middleware for General-Purpose Computations, *Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.

F. Chang, V. Karamcheti, and Z. Kedem, Exploiting Application Tunability for Efficient, Predictable Parallel Resource Management , *International Parallel Processing Symposium (IPPS'99)*, April 1999.

A. Baratloo, A. Itzkovitz, Z. Kedem, and Y. Zhao, Mechanism for Just-in-time Allocation of Resources to Adaptive Parallel Programs, *Proceedings of the International Parallel Processing Symposium (IPPS/SPDP 1999)*, April 1999.

I. Lipkind and V. Karamcheti, Object Views: Bridging the Performance Gap between Shared Memory and Message Passing, *Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.

I. Lipkind, I. Pechtchanski, and V. Karamcheti, Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations, *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, November 1999.